Introduction to C language

C is a computer high level language available on the UNIX operating systems. C lets you write your programs clearly and it has decent control flow facilities so your code can be read straight down the page. It lets you write structured code that is compact without being too cryptic; it encourages modularity and good program organization; and it provides good data-structuring facilities.
Creating the program

- Create a file containing the complete program. You can use any ordinary editor with which you are familiar to create the file. One such editor is ***textedit*** , **VI editor** available on most UNIX systems.
- The filename must by convention end ``.c'' (full stop, lower case c), ***e.g. firstprog.c*** . (file name is user defined). The contents must obey C syntax.

A First Program

```
#include < stdio.h>

void main()
{
   printf("\nThis is my first program in C\n");
}
```

- Save the code in the file ***`firstprog.c`***
- A C program contains *functions* and *variables*. The functions specify the tasks to be performed by the program. The ``main'' function establishes the overall logic of the code. It is normally kept short and calls different functions to perform the necessary sub-tasks. All C codes must have a ``main'' function.
- Program ***`firstprog.c`*** code calls ***`printf`***, an output function from the I/O (input/output) library (defined in the file ***`stdio.h`***).
- The original C language did not have any built-in I/O statements whatsoever. Nor did it have much arithmetic functionality. The original language was really not intended for "scientific" or "technical" computation.. These functions are now performed by standard libraries, which are now part of ANSI C.
- The ***`printf`*** line prints the message **`This is my first program in C`** on ``stdout'' (the output stream corresponding to the X-terminal window in which you run the code); ``\n'' prints a ``new line'' character, which brings the cursor onto the next line. By construction, `printf` never inserts this character on its own: the following program would produce the same result:

```
#include < stdio.h>

void main()
{
   printf("\n");
   printf("This is my first program in C");
   printf("\n");
}
```

- The first statement ``***`#include < stdio.h>`***'' includes a specification of the C language I/O library.
- All variables in C must be explicitly defined before use: the ``***`.h`***'' files are by convention ``header files'' which contain definitions of variables and functions necessary for the functioning of a program, whether it be in a user-written section of code, or as part of the standard C libaries.
- The directive ``#include'' tells the C compiler to insert the contents of the specified file at that point in C code. The ``< ...>'' notation instructs the compiler to look for the file in certain ``standard'' system directories.
- The ***`void`*** preceding ``main'' indicates that main is of ``void'' type i.e., it has ***no*** type associated with it, meaning that it cannot return a result on execution.
- The ``;'' denotes the end of a statement.
- Blocks of statements are put in braces {...}, as in the definition of functions.
- All C statements are defined in free format, i.e., with no specified layout or column assignment.
- Whitespace (tabs or spaces) is never significant, except inside quotes as part of a character string.

Compilation

- There are many C compilers around. The ***`cc`*** being the default Sun compiler. The ***gcc*** compiler is popular and available for many platforms.
- Save C code in the file ***`firstprog.c`***, then compile it by typing at unix prompt: ***cc firstprog.c*** or ***gcc firstprog.c***. This creates an *executable* file ***`a.out`***, which is then executed simply by typing its name.
- If there are syntax errors (compilation errors) in your program (such as mistypings, misspelling one of the key words or omitting a semi-colon), the compiler will detect and report them.
- There may, of course, still be logical errors that the compiler cannot detect. You may be telling the computer to do the wrong operations such as division by 0 etc..
- When the compiler has successfully compiled your program, the compiled or executable file is stored in a default file called ***a.out***.

- If you want to store executable file in user defined file, then the following option is used which puts the compiled program into the file ***anyname*** instead of putting it in the file **a.out**.

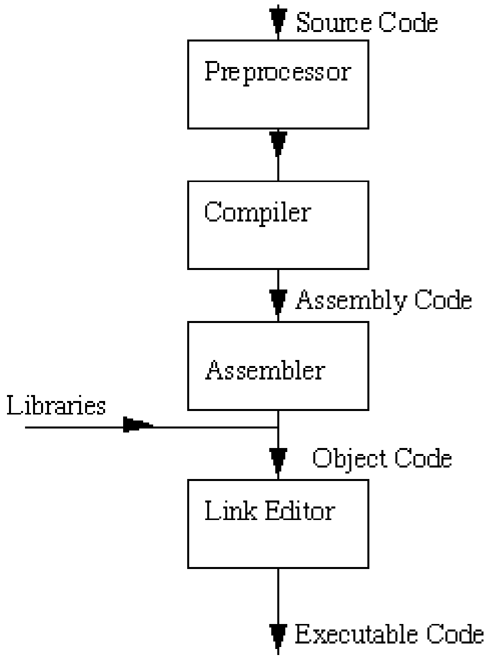
```
cc -o anyname firstprog.c
```

Running the program

- The next stage is to actually run your executable program. To run an executable in UNIX, you simply type the name of the file containing it, in this case ***anyname*** (or ***a.out***)
- This executes your program, printing any results to the screen. At this stage there may be run-time errors, such as division by zero, or it may become evident that the program has produced incorrect output.
- If so, you must return to edit your program source, and recompile it, and run it again.

The C Compilation Model

Following are the key features of the C Compilation model



The Preprocessor

- The Preprocessor accepts source code as input and is responsible for removing ***comments*** and interpreting special ***preprocessor directives*** denoted by `#`.

For example

`#include` -- includes contents of a named file. Files usually called ***header*** files. ***e.g***

`#include <math.h>` -- standard library maths file.

`#include <stdio.h>` -- standard library I/O file

`#define` -- defines a symbolic name or constant. Macro substitution.

```
#define MAX_ARRAY_SIZE 100
```

C Compiler

The C compiler translates source to assembly code. The source code is received from the preprocessor.

C Program Structure

A C program basically has the following components:

- Preprocessor Commands
- Type definitions
- Function prototypes -- declare function types and variables passed to function.
- Variables
- Functions definitions
- main( )or void main( ) function. Some compilers do not accept main( )
- C requires a semicolon at the end of **every** statement.

Overall structure of C program is as follows:

main()
{ **declarations**
-- variables, arrays, records, function declarations etc each one separated out by semi colon}
statements
-- each separated out by semi colon
}
Function definitions

Ex:

```
#include <stdio.h>
#define i 6
void main()
{       int x, y;
        x=7;  y= i + x;
        printf("%d\n", y);
```

}

A function has the form:

```
type function_name (parameters)
      {
             local variables
             C Statements

       }
```

- If the type definition is omitted C assumes that function returns an **integer** type. **NOTE:** This can be a source of problems in a program.

Variables

C has the following simple data types:

C type	Size (bytes)	Lower bound	Upper bound
char	1	—	—
unsigned char	1	0	255
short int	2	-32768	$+32767$
unsigned short int	2	0	65536
(long) int	4	-2^{31}	$+2^{31}-1$
float	4	$-3.2 \times 10^{\pm 38}$	$+3.2 \times 10^{\pm 38}$
double	8	$-1.7 \times 10^{\pm 308}$	$+1.7 \times 10^{\pm 308}$

- On UNIX systems all ***ints*** are `long ints` unless specified as `short int` explicitly.
- **NOTE:** There is **NO** Boolean type in C
    - either use `char, int` or (better) `unsigned char`.
    - `Unsigned` can be used with all `char` and `int` types.

- To declare a variable in C, write as follows:
  `var_type` ***list variables***;

```
int i,j,k;
float x,y,z;
char ch;
```

Defining Global Variables

- Global variables are defined above `main()` in the following way:-

```
      short number,sum;
      int bignumber,bigsum;
      char letter;

       main()
       {
            body
       }
```

- It is also possible to pre-initialize global variables using the `= operator for assignment.`

```
      float sum=0.0;
      int bigsum=0;
      char letter=`A';
      main()
      {

      }
```

- This is the same as but initialization is more efficient.

```
      float sum;
      int bigsum;
      char letter;
      main()
      {
            sum=0.0;
            bigsum=0;
            letter=`A';

      }
```

- C also allows multiple assignment statements using =, such as a=b=c=d=3; which is the same as having separate assignment to each variable, but is more efficient.
- This kind of assignment is only possible if all the variable types in the statement are the same.
- You can define your own data types. As an example of a simple use let us consider how we may define two new types real and letter. These new types can then be used in the same way as the pre-defined C types:

```
      typedef real float;
      typedef letter char;
```

Variables declared:

```
       real sum=0.0;
       letter nextletter;
```

Printing Out and Inputting Variables

- C uses formatted output. The ***`printf`*** function has a special formatting character (%) -- a character following this defines a certain format for a variable:

```
%c – characters
%d – integers
%f -- floats
```

Example: `printf(“%c %d %f”,ch,i,x);`

- Format statement is enclosed in “...”, variables follow after. Make sure order of format and variable data types match up.

- `scanf()` is the function for inputting values to a data structure:
- Its format is similar to `printf`: ***i.e.*** `scanf(“%c %d %f”,&ch,&i,&x);`
- Here & before variables means address of the variables. Variables appearing in scanf always are to be preceded by &.

Constants

- ANSI C allows you to declare ***constants***. When you declare a constant it is a bit like a variable declaration except the value cannot be changed.
- The ***`const`*** keyword is to declare a constant, as shown below:

```
int const a = 1;
const int a =2;
```

- You can declare the ***`const`*** before or after the **typedef**.
- The preprocessor `#define` is another more flexible method to define ***constants*** in a program.
- You frequently see const declaration in function parameters. This says simply that the function is **not** going to change the value of the parameter.

Operations

Arithmetic operators

- Standard arithmetic operators (`+ - * /`) are found in most languages, C provides some more operators.
- Assignment is = ***i.e.*** *i* = 4; `ch` = `y';
- Increment `++`, Decrement `--` which are more efficient than their long hand equivalents, for example: `x++` is faster than `x=x+1`.

- The `++` and `--` operators can be either in post-fixed or pre-fixed. With pre-fixed the value is computed before the expression is evaluated whereas with post-fixed the value is computed after the expression is evaluated.
- The % (modulus) operator only works with integers.
- Division / is for both integer and float division. So be careful.
- The answer to: *x* = 3 / 2 is 1 even if *x* is declared a float!!
- If both arguments of / are integer then do integer division.

Comparison Operators

- To test for equality use double equality symbol = = (if (x = = y) …
- Beware of using ``=" instead of ``= =", such as writing accidentally

```
        if ( i = j ) .....
```

- This is a perfectly **LEGAL** C statement (syntactically speaking) which copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if `j` is non-zero.
- Not equals is: !=
- Other operators < (less than) , > (grater than), <= (less than or equals), >= (greater than or equals) are as usual.

Logical Operators

- Logical operators are usually used with conditional statements which we shall meet in the next Chapter.
- The two basic logical operators are:
    - && for logical AND,
    - || for logical OR.
- The operators & and | have a different meaning for bitwise AND and OR

Order of Precedence

- It is necessary to be careful of the meaning of such expressions as `a + b * c`
- We may want the effect as either

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

- All operators have a priority, and high priority operators are evaluated before lower priority ones. Operators of the same priority are evaluated from left to right, so that `a - b - c` is evaluated as `(a - b ) - c`
- From high priority to low priority the order for all C operators is:

```
        ( )  [  ]  -> .
         !  ~  - * & sizeof cast ++ -
               (these are right->left)
         * / %
         + -
         < <= >= >
         == !=
         &
```

```
^          |
&&
||
?:   (right->left)
= += -= (right->left)
,   (comma)
```

- Thus `a < 10 && 2 * b < c` is interpreted as
  `( a < 10 ) && ( ( 2 * b ) < c )` and

Conditionals Statements

- Various methods that C can control the ***flow*** of logic in a program are given in this section.
- Apart from slight syntactic variation they are similar to other languages.
- As we have seen following logical operations exist in C:
  = =, !=, ǁ , &&.
- One other operator is the unary operator. It takes only one argument i.e., ***not*** **!**.
- These operators are used in conjunction with the following statements.

The `if` statement

The `if` statement has the same function as other languages. It has three basic forms:

- `if (`*expression*`)` *statement*
- `if (`*expression*`)` *statement*$_1$
  `else` *statement*$_2$
- `if (`*expression*`)` *statement*$_1$
  `else if (`*expression*`)` *statement*$_2$
  `else` *statement*$_3$

}

The `?` operator

The `?` (***ternary condition***) operator is a more efficient form for expressing simple conditional expression. It has the following form:

expression$_1$ `?` *expression*$_2$`:` *expression*$_3$

- It simply states: `if` *expression*$_1$ `then` *expression*$_2$ `else` *expression*$_3$
- For example to assign the maximum of `a` and `b` to `z`: `z = (a>b) ? a : b;` which is the same as:

```
if (a>b) z = a;
 else z=b;
```

The `switch` statement

The C ***switch*** allows multiple choice of a selection of items at one level of a conditional where it is a far neater way of writing multiple `if` statements:

```
switch (expression)
{
      case item_1:       statement_1;
                         break;
      case item_2:       statement_2;
                         break;

            ¦
            ¦
      case item_n:       statement_n;
                         break;
      default:           statement;
                         break;
}
```

- In each case the value of *item*$_i$ must be a constant and variables are <u>not</u> allowed.
- The ***break*** is needed if you want to terminate the ***switch*** after execution of one choice. Otherwise the next case would get evaluated.
- We can also have **null** statements by just including a **`:`** or let the switch statement ***fall through*** by omitting any statements (see ***e.g.*** below).
- The ***default*** case is optional and catches any other cases.

For example:-

```
          switch (letter)
            {
                  case `A':
                  case `E':
                  case `I':
                  case `O':
                  case `U':   numberofvowels++;
                              break;

                  case ` ':   numberofspaces++;
                              break;

                  default:    numberofconstants++;
                              break;
            }
```

- In the above example if the value of `letter is `A', `E', `I', `O' or `U' then numberofvowels is incremented.`
- `If the value of letter is ` ' then numberofspaces is incremented.`
- `If none of these is true then the default condition is executed, that is numberofconstants is incremented.`

The `for` statement

The C ***for*** statement has the following form:

```
for  (expression1; expression2; expression3)
     statement; or {block of statements}
```

- *expression$_1$* initializes;
- *expression$_2$* is the terminate test;
- *expression$_3$* is the modifier (which may be more than just simple increment);
- C basically treats `for` statements as `while` type loops

The `while` statement

- The `while` has the form:

```
while (expression) statement
```

For example:

```
int x=3;
main()
{
     while (x>0)
     {     printf("x=%d\n",x);
           x--;
     }
}
```

...outputs to the screen:

```
x=3
x=2
x=1
```

- Since the while loop can accept expressions, not just conditions, the following are all legal:-
  - o while (x--);
  - o while (x=x+1);
  - o while (x+=5);
- Using this type of expression, only when the result of x--, x=x+1, or x+=5, evaluates to 0 will the while condition fails and the loop be exited.
- We can go further still and perform complete operations within the while ***expression***:
  - o while (i++ < 10); It counts i up to 10
  - o while ((ch = getchar()) != `q') putchar(ch);
- In this example C uses standard library functions getchar() that reads a character from the keyboard and putchar() writes a given char to screen.
- The while loop will proceed to read from the keyboard and echo characters to the screen until a 'q' character is read.

The `do-while` statement

C `do-while` statement has the form:

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

For example:

```
int x=3;

main()
{     do
      {     printf("x=%d\n",x--);
                          }
      while (x>0);
      }
}
```

- It outputs:-

```
x=3
x=2
x=1
```

NOTE: The postfix `x--` operator which uses the current value of `x` while printing and ***then*** decrements `x`.

`break` and `continue`

C provides two commands to control how we loop:

- `break` -- exit form loop or switch.
- `continue` -- skip 1 iteration of loop.

Consider the following example where we read in integer values and process them according to the following conditions. If the value we have read is negative, we wish to print an error message and abandon the loop. If the value read is great than 100, we wish to ignore it and continue to the next value in the data. If the value is zero, we wish to terminate the loop.

```
while (scanf( ``%d'', &value ) == 1 && value != 0)
{

      if (value < 0)
      {     printf(``Illegal valuen'');
            break;      /* Abandon the loop */
      };
      if (value > 100)
      {     printf(``Invalid valuen'');
            continue;  /* Skip to start loop again */
      };
}
```

Arrays and Strings

- In principle, arrays in C are similar to those found in other languages. As we shall shortly see arrays are defined slightly differently and there are many subtle differences due the close link between array and pointers.

Single and Multi-dimensional Arrays

- Arrays in C are defined as: `int listofnumbers[50];`
- In C Array subscripts start at **0** and end one less than the array size whereas in other languages like fortran, pascal it starts from 1.
- For example, in the above case valid subscripts range from 0 to 49.
- Elements can be accessed in the following ways:-

```
            thirdnumber=listofnumbers[2];
            listofnumbers[5]=100;
```

- Multi-dimensional arrays can be defined as follows:

```
            int tableofnumbers[50][50]; for two dimensions.
```

- For further dimensions simply add more [ ]:

```
      int bigD[50][50][40][30]......[50];
```

- Elements can be accessed in the following ways:

```
            anumber=tableofnumbers[2][3];
            tableofnumbers[25][16]=100;
```

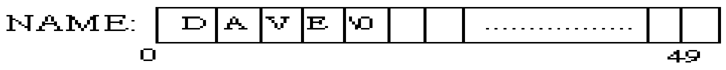
Strings

- In C, Strings are defined as arrays of characters. For example, the following defines a string of 50 characters: `char name[50];`
- C has no string handling facilities built in and so the following assignments are illegal:

```
      char firstname[50],lastname[50],fullname[100];

       firstname= "Arnold"; /* Illegal */
       lastname= "Schwarznegger"; /* Illegal */
       fullname= "Mr"+firstname +lastname; /* Illegal */
```

- However, there is a special library of string handling routines which may be included in header file and then various string operations can be used.
- To print a string we use printf with a special **%s** control character: `printf(``%s'',name);`
- **NOTE:** We just need to give the name of the string.
- In order to allow variable length strings the 0 character is used to indicate the end of a string.

- So we if we have a string, char NAME[50]; and we store the ``DAVE'' in it its contents will look like:



Functions

- C provides functions which are again similar in most languages. One difference is that C regards **`main()`** as a function.
- The form of a C function is as follows:

```
returntype fn_name(parameterdef1, parameterdef2,---)

      {

             localvariables

             functioncode

      }
```

- It returns the value of *`returntype.`*
- Let us look at an example to find the average of two integers:

```
float findaverage(float a, float b)
{     float average;
      average=(a+b)/2;
      return(average);
}
```

- We would ***call*** the function as follows:

```
 main()
{     float a=5,b=15,result;
      result=findaverage(a,b);
      printf("average=%f\n",result);
}
```

- The return statement passes the result back to the main program.

`void` functions

- The `void` function provides a way of not returning any value through function name but if needed, values can be returned using variables in parameter list.
- Here `return` statement is not used:

```
void squares()
{     int loop;
      for (loop=1;loop<10;loop++);
      printf("%d\n",loop*loop);
}
```

- In the main function we can call it as follows:

```
main()

{     squares(); }
```

- We must have ( ) even for no parameters unlike some languages.

Functions and Arrays

- Single dimensional arrays can be passed to functions as follows:-

```
float findaverage(int size,float list[])
{     int i; float sum=0.0;
      for (i=0;i<size;i++) sum+=list[i];
      return(sum/size);
}
```

- Here the declaration `float list[]` tells C compiler that **list** is an array of float type. It should be noted that dimension of array is not specified when it is a ***parameter*** of a function.
- Multi-dimensional arrays can be passed to functions as follows:

```
void printtable(int xsize,int ysize, float table[][5])
{     int x,y;
      for (x=0; x<xsize; x++)
            {     for (y=0; y<ysize;y++)
                  printf("\t%f",table[x][y]);
                  printf("\n");
            }
}
```

- Here float table[][5] tells C compiler that **table** is an array of dimension N X 5 of float. Note we must specify the second (and subsequent) dimension of the array BUT not the first dimension.

Function Prototyping

- Before you use a function, C must have ***knowledge*** about the type it returns and the parameter types the function expects.
- The ANSI standard of C introduced a new (better) way of doing this than previous versions of C. (Note: All new versions of C now adhere to the ANSI standard.)
- The importance of prototyping is twofold.
- It makes for more structured and therefore easier to read code.
- It allows the C compiler to check the ***syntax*** of function calls.

- How this is done depends on the scope of the function. Basically if a functions has been defined before it is used (called) then you are ok to merely use the function.
- If NOT then you must ***declare*** the function. The declaration simply means the header line of function declaration i.e., the type the function returns and the type of parameters used by the function ***e.g.*** `int strlen(char []);`
- It is usual (and therefore **good**) practice to prototype all functions at the start of the program, although this is not strictly necessary.

Further Data Types

Here we discuss more advanced data types and structures used in a C program.

Structures

The main use of structures is to lump together collections of disparate variable types, so they can conveniently be treated as a unit. For example:

```
struct employee
      {
            char name[50];
            char sex;
            float salary;
      };
```

- **struct empolyee xyz;** defines a new structure **gun** and makes **xyz** an instance of it.
- Here gun is a ***tag*** for the structure that serves as shorthand for future declarations. We now only need to say **struct gun** and the body of the structure is implied as we do to make the xyz variable. The tag is ***optional***.
- Variables can also be declared between the } and ; of a struct declaration, ***i.e.***:

```
struct employee
      {
            char name[50];
            char sex;
            float salary;
      } xyz;
```

- struct can be pre-initialized at declaration: **struct xyz ={"john", ’m’, 20000.50}**;
- To access a member (or field) of a struct, C provides dot (**.**) operator.
- For example, xyz **.** sex ; xyz **.** salary; xyz **.** name

Defining New Data Types

- ***typedef*** can also be used with structures. The following creates a new type emp_type which is of type **struct** employee and can be initialized as usual:

```
typedef struct employee
      {
            char name[50];
            char sex;
            float salary;
      } emp_type xyz ={"john", 'm', 20000.50};
```

- Here employee still acts as a ***tag*** to the struct and is optional.
- Indeed since we have defined a new data type it is not really of much use,
- Emp_type is the new data type. xyz is a variable of type emp_type which is a structure.
- C also allows arrays of structures:

```
typedef struct employee
      {
            char name[50];
            char sex;
            float salary;
      } emp_type;

emp_type emp[100];
```

- Here emp is an array of 100 elements with each element of type emp_type.

Unions

- An union is a variable which may hold (at different times) objects of different sizes and types. C uses the union statement to create unions, for example:

```
union number
        {
              short shortnumber;
              long longnumber;
              double floatnumber;
        } anumber
```

- It defines a union called number and an instance of it called anumber. **number** is a union ***tag*** and acts in the same way as a tag for a structure.
- Members can be accessed in the following way:

```
printf("%ld\n",anumber.longnumber);
```

- This clearly displays the value of longnumber.
- When C compiler is allocating memory for unions it will always reserve enough room for the largest member (in the above example this is 8 bytes for the double).
- In order that the program can keep track of the type of union variable being used at a given time it is common to have a structure (with union embedded in it) and a variable which flags the union type.
- For example:

```
typedef struct
      {
            int maxpassengers;
      } jet;

typedef struct
      {
            int liftcapacity;
      } helicopter;

typedef struct
      {
            int maxpayload;
      } cargoplane;

typedef union
      {
            jet jetu;
            helicopter helicopteru;
            cargoplane cargoplaneu;
      } aircraft;

typedef struct
      {
            aircrafttype kind;
            int speed;
            aircraft description;
      } an_aircraft;
```

- This example defines a base union aircraft which may either be jet,helicopter,or cargoplane.
- In the an_aircraft structure there is a kind member which indicates which structure is being held at the time.

Coercion or Type-Casting

- C is one of the few languages to allow ***coercion***, that is forcing one variable of one type to be another type.
- C allows this using the cast operator `()`.

```
int integernumber;
float floatnumber=9.87;

integernumber=(int)floatnumber;
```

- It assigns 9 (the fractional part is thrown away) to `integernumber`.

```
      int integernumber=10;
      float floatnumber;

      floatnumber=(float)integernumber;
```

- It assigns 10.0 to floatnumber.
- Coercion can be used with any of the simple data types including char. For example:

```
      int integernumber;
      char letter='A';

      integernumber=(int)letter;
```

- It assigns 65 (the ASCII code for `A') to integernumber.
- Some typecasting is done automatically -- this is mainly with integer compatibility.
- A good rule to follow is: **If in doubt cast**.
- Another use is the make sure division behaves as requested: If we have two integers internumber and anotherint and we want the answer to be a float then

```
      floatnumber = (float) internumber / (float) anotherint;
```

- It ensures floating point division.

Enumerated Types

- Enumerated types contain a list of constants that can be addressed in integer values.
- We can declare types and variables as follows.

```
            enum days {monday, tuesday, ..., sunday} week;
            enum days week1, week2;
```

- As with arrays first enumerated name has index value 0. So `monday` has value 0, `tuesday` 1, and so on. **`week1`** and **`week2`** are variables. We can define other values:

```
            enum escapes { bell = `\a',
                  backspace = `\b',  tab = `\t',
                  newline = `\n', vtab = `\v',
                  return = `\r'};
```

- We can also override the 0 start value:

```
      enum months {jan = 1, feb, mar, ......, dec};
```

- Here it is implied that feb = 2 ***etc.***

Static Variables

- A **`static`** variable is <u>local</u> to particular function. However, it is only initialized once (on the first call to function).
- Also the value of the variable on leaving the function remains **intact**. On the next call to the function the the `static` variable has the same value as on leaving.
- To define a `static` variable simply prefix the variable declaration with the `static` keyword. For example:

```
void stat(); /* prototype fn */

main()
  {   int i;
      for (i=0;i<5;++i)  stat();
  }

void stat()
  {   int auto_var = 0;
      static int static_var = 0;
printf(“auto = %d, static = %d\n”, auto_var, static_var);
      ++auto_var;
      ++static_var;
  }
```

Output is:

```
auto_var = 0, static_var= 0
auto_var = 0, static_var = 1
auto_var = 0, static_var = 2
auto_var = 0, static_var = 3
auto_var = 0, static_var = 4
```

- Clearly the **`auto_var`** variable is created each time. The **`static_var`** is created once and remembers its value.

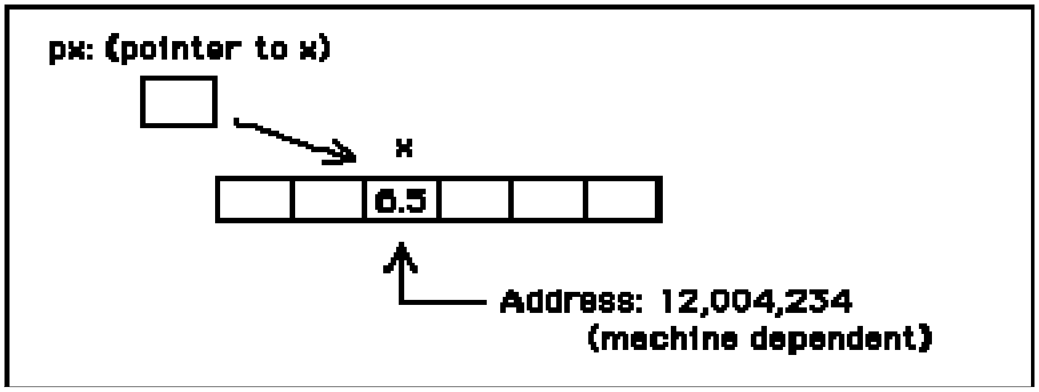
Pointers

- Pointer is a fundamental part of C. If you cannot use pointers properly then you have basically lost all the power and flexibility that C allows.
- The secret to C is in its use of pointers.
- C uses pointers a lot.
  - It is the only way to express some computations.
  - It produces compact and efficient code.
  - It provides a very powerful tool.
- C uses pointers explicitly with:
  - Arrays,
  - Structures,
  - Functions.

What is a Pointer?

- A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type.
- The ***unary*** or ***monadic*** operator **&** gives the ``address of a variable".
- The ***indirection*** or dereference operator ***** gives the contents of an object ***pointed to*** by a pointer.
- To declare a pointer to a variable do: **`int *pointer;`**
- In the following example `px` is a pointer to objects of type float, and sets it equal to the address of `x`:

```
float x;
float* px;
x = 6.5;
px = &x;
```

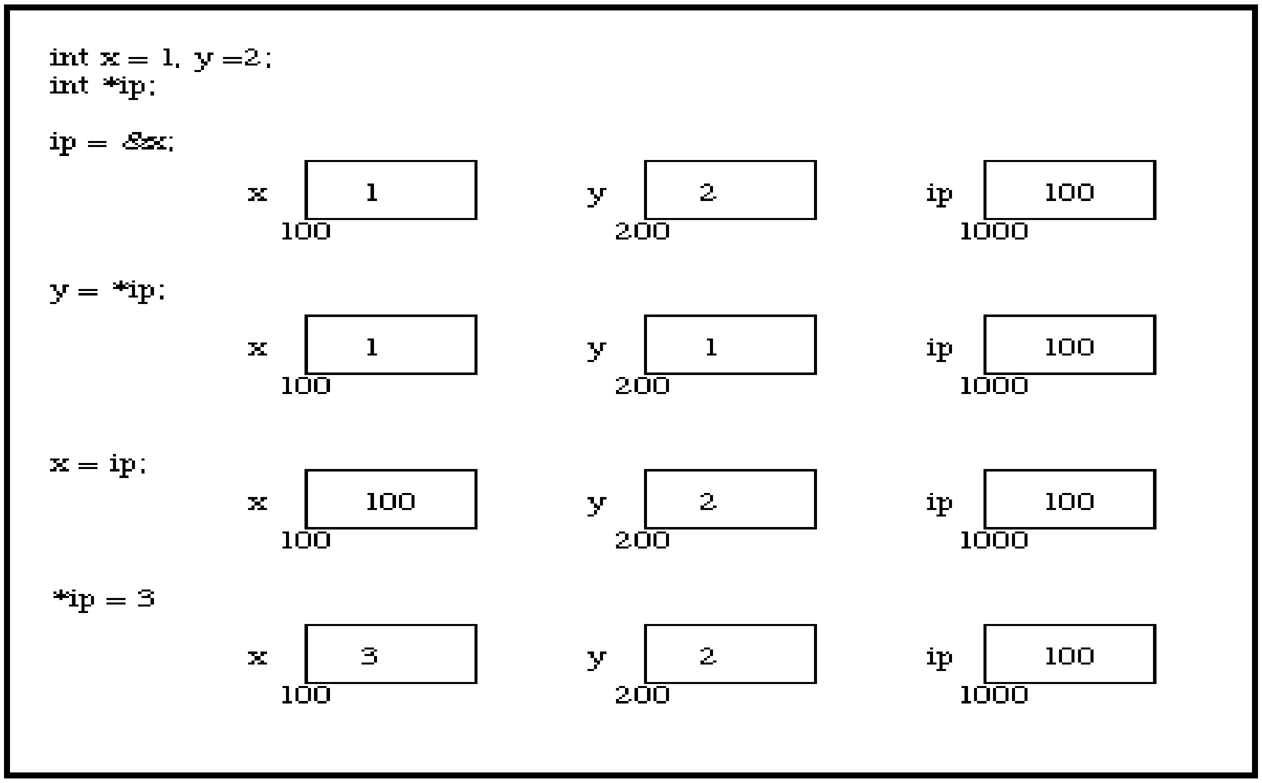


- The content of the memory location referenced by a pointer is obtained using the ``*" operator (this is called ***dereferencing*** the pointer). Thus, `*px` refers to the value of `x`.

- We must associate a pointer to a particular type: You can't assign the address of a **short int** to a **long int**. For example, cnsider the effect of the following code:

```
int x = 1, y = 2;
int *ip;

ip = &x;
y = *ip;
x = ip;
*ip = 3;
```

- It is worth considering what is going on at the ***machine level*** in memory to fully understand how pointer works. Consider the following figure. Assume for the sake of this discussion that variable x resides at memory location 100, y at 200 and ip at 1000. Effect of each of above four statements is shown separately



Pointer, Variables and Memory

- Now the assignments x = 1 and y = 2 obviously load these values into the variables. ip is declared to be a ***pointer to an integer*** and is assigned to the address of x (&x). So ip gets loaded with the value 100.
- Next y gets assigned to the ***contents of*** ip. In this example ip currently ***points*** to memory location 100 (the location of x). So y gets assigned to the values of x which is 1.
- We have already seen that C is not too fussy about assigning values of different type. Thus it is perfectly **legal** (although not all that common) to assign the current value of ip to x. The value of ip at this instant is 100.
- Finally we can assign a value to the contents of a pointer (*ip).
- When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it. So the following statements will generate an error (program crash!!).

```
int *ip;
*ip = 100;
```

- The correct use is:

```
int *ip;
int x;

ip = &x;
*ip = 100;
```

- We can do integer arithmetic on a pointer:

```
float *p, *q;
*p = *p + 10;
 ++*p;
 (*p)++;
 q = p;
```

- A pointer to any variable type is an address in memory which is an integer address. A pointer is definitely NOT an integer.
- The reason we associate a pointer to a data type is so that it knows how many bytes the data is stored in. When we increment a pointer we increase the pointer by one ``block'' memory.

Pointer and Functions

- Let us now examine the close relationship between pointers and C's other major parts. We will start with functions.
- When C passes arguments to functions it passes them by value.
- There are many cases when we may want to alter a passed argument in the function and receive the new value back once to function has finished. C uses pointers explicitly to do this.
- The best way to study this is to look at an example where we must be able to receive changed parameters.
- Let us try and write a function to swap variables around?
- The usual function ***call***: `swap(a, b)` WON'T WORK.
- Pointers provide the solution: ***Pass the address of the variables to the functions and access address of function.***
- Thus our function call in our program would look like this: `swap(&a, &b)`
- The Code to swap is fairly straightforward:

```
void swap(int *px, int *py)
{     int temp;
      temp = *px;
/* contents of pointer */
      *px = *py;
      *py = temp;
 }
```

- We can return pointer from functions. A common example is when passing back structures. ***e.g.***:

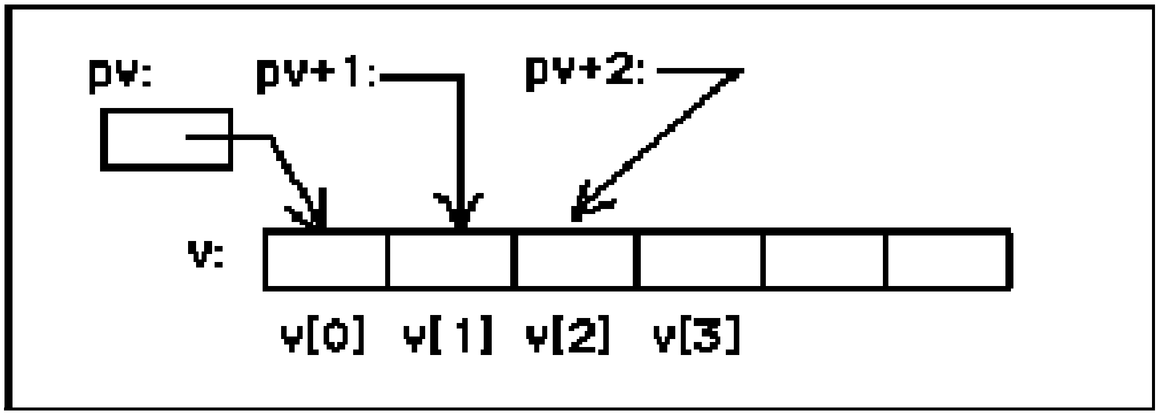
```
typedef struct
            {           float x,y,z;
            } COORD;

 main()

  {        COORD p1,
           *coord_fn(); /* declare fn to return ptr of
                                         COORD type */
            ....
            p1 = *coord_fn(...); /* assign contents of address
                                                returned */
            ....
 }

COORD *coord_fn(...)
{          COORD p;


            .....
           p = ....;             /* assign structure values */

           return &p;                   /* return address of p */
}
```

- Here we return a pointer whose contents are immediately ***unwrapped*** into a variable.
- We must do this straight away as the variable we pointed to was local to a function that has now finished.
- This means that the address space is free and can be overwritten.
- It will not have been overwritten straight after the function has quit though so this is perfectly safe.

Pointers and Arrays

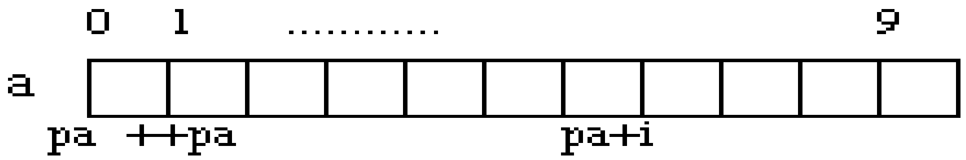
- Arrays of any type can be formed in C. The syntax is simple: **type name[dim];**
- In C, array starts at position 0. The elements of the array occupy adjacent locations in memory.
- C treats the name of the array as if it were a pointer to the first element. This is important in understanding how to do arithmetic with arrays. Thus, if `v` is an

array, `*v` is the same thing as `v[0]`, `*(v+1)` is the same thing as `v[1]`, and so on: here pv is simply indication pointer v.



- Pointers and arrays are very closely linked in C. To make it more clear let us see another example.

```
int a[10], x;
int *pa;
pa = &a[0];  /* pa pointer to address of a[0] */
x = *pa;  /* x = contents of pa (a[0] in this case) */
pa + i  ≅  a[i]  /*  ≅ means equivalent */
```



- There is no bound checking of arrays and pointers so you can easily go beyond array memory and overwrite other things.
- C however is much more subtle in its link between arrays and pointers.
- For example we can just type **pa = a;** instead of **pa = &a[0]** and a[i] can be written as *(a + i) ***i.e.*** &a[i] ≡a + i.
- We also express pointer addressing like this: pa[i] ≅ *(pa + i).
- However pointers and arrays are different:
  - A pointer is a variable. We can do pa = a and pa++.
  - An Array <u>is not</u> a variable. a = pa and a++ ARE ILLEGAL.

- We can now understand how arrays are passed to functions.

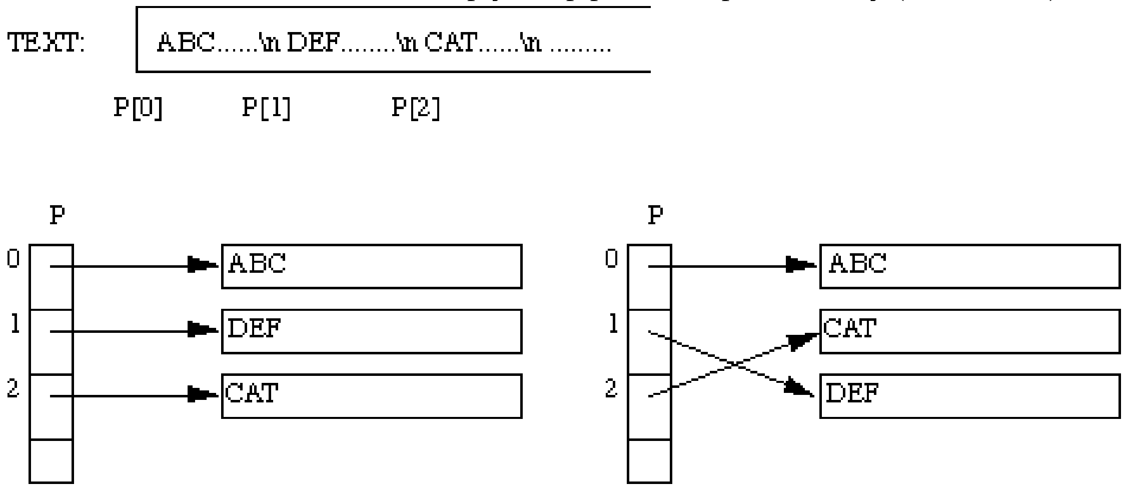
- When an array is passed to a function what is actually passed is its initial elements location in memory. So:

  strlen(s) ≅ strlen(&s[0])
- This is why we declare the function: int strlen(char s[]);
- An equivalent declaration is : int strlen(char *s); since char s[] ≅ char *s.
- strlen() is a ***standard library*** function that returns the length of a string. Let's look at how we may write a function:

```
int strlen(char *s)
{       char *p = s;

        while (*p != `\0);
        p++;
        return p-s;
}
```

Arrays of Pointers

- We can have arrays of pointers since pointers are variables.
- ***Arrays of Pointers*** are a data representation that will cope efficiently and conveniently with variable length text lines. How can we do this?:
- Store lines end-to-end in one big `char` array. `\n` will delimit lines.
- Store pointers in a different array where each pointer points to 1st char of each new line.
- Compare two lines using `strcmp()` standard library function.
- If 2 lines are out of order simply swap pointer in pointer array (not the text).



- This eliminates complicated storage management and high overheads of moving lines.

Multidimensional arrays and pointers

- We should think of multidimensional arrays in a different way in C:
- A 2D (two dimensional) array (in maths similar to matrix) is really a 1D array, each of whose elements is itself an array.
- Hence in `a[n][m]` notation, there are n rows and m columns, array elements are stored row by row.
- When we pass a 2D array to a function we must specify the number of columns and the number of rows is irrelevant.
- The reason for this is pointers again. C needs to know how many columns in order that it can jump from row to row in memory.
- Consider `int a[5][35]` to be passed in a function:
    - We can do:

      ```
      f(int a[][35]) {.....}
      ```

      or even:

      ```
      f(int (*a)[35]) {.....}
      ```
- We need parenthesis (*a) since [] have a higher precedence than * . So:

  ```
  int (*a)[35]; /* declares a pointer to an array of 35 ints.*/
  int *a[35];   /* declares an array of 35 pointers to ints.  */
  ```
- Now let us look at the (subtle) difference between pointers and arrays. Strings are a common application of this.
- Consider:

  ```
  char *name[10];
  char Aname[10][20];
  ```
- `name` has 10 pointer elements.
- `Aname` is a 2 D char array with 200 elements.
- An element access is made in memory by using following formula.

  20*row + col + base_address

Static Initialisation of Pointer Arrays

- Initialization of arrays of pointers is an ideal application for an internal static array.

  ```
  some_fn( )
  {       static char *months =
                  { “no month”,”jan”, “feb”, ...};

  }
  ```
- **static** reserves a private permanent bit of memory.

Pointers and Structures

- These are fairly straight forward and are easily defined. Consider the following:

  ```
  struct COORD
        {
              float x,y,z;
  ```

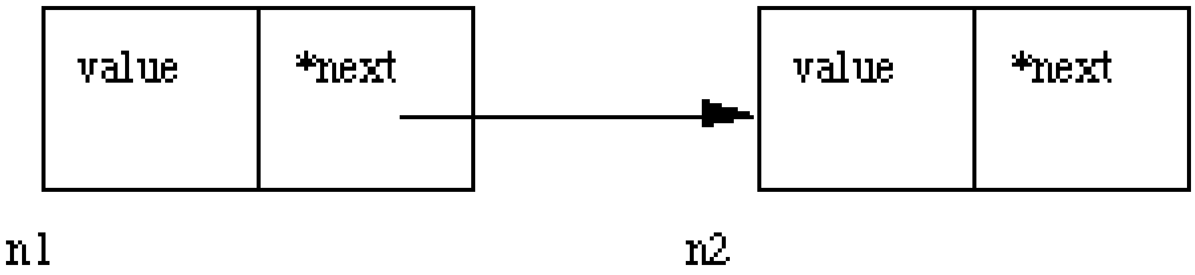
```
      } pt;
struct COORD *pt_ptr;
pt_ptr = &pt; /* assigns pointer to pt */
```

- The operator → lets us access a member of the structure pointed to by a pointer.***i.e.***:

```
pt_ptr →  x = 1.0;
pt_ptr → y = pt_ptr → y - 3.0;
```

- Example: Linked Lists

```
typedef struct
      {     int value;
            ELEMENT *next;
      } ELEMENT;

ELEMENT n1, n2;
n1.next = &n2;
```



- We can only declare `next as a pointer to ELEMENT. We cannot have a element of the variable type as this would set up a` ***recursive*** `definition which is` **`NOT ALLOWED`**`.`
- `We are allowed to set a pointer reference since 4 bytes are set aside for any pointer. The above code links a node n1 to n2`

Common Pointer Pitfalls

- Here we will highlight two common mistakes made with pointers.
  - Not assigning a pointer to memory address before using it

```
int *x;
*x = 100;
```

  - `we need a physical location say: int y;`

```
x = &y;
*x = 100;
```

  - This may be hard to spot. **NO COMPILER ERROR**. Also x could get some random address at initialization.

Dynamic Memory Allocation and Dynamic Structures

- Dynamic allocation is a pretty unique feature to C (amongst high level languages).
- It enables us to create data types and structures of any size and length to suit our programs need within the program.
- There are two common applications of this:
  - dynamic arrays
  - dynamic data structure ***e.g.*** linked lists

Malloc, Sizeof, and Free

- The system defined function **`malloc`** is most commonly used to attempt to ``grab'' a continuous portion of memory.
- It is defined by: **`void *malloc(size_t number_of_bytes)`**
- It says that it returns a pointer of type `void *` that is the start in memory of the reserved portion of size `number_of_bytes`.
- If memory cannot be allocated a `NULL` pointer is returned.
- Since a `void *` is returned, the C standard states that this pointer can be converted to any type.
- The `size_t` argument type is defined in `stdlib.h` and is an ***unsigned type***. So:

```
char *cp;
cp = malloc(100);
```

- It attempts to get 100 bytes and assigns the start address to `cp.`
- Also it is usual to use the sizeof() function to specify the number of bytes:

```
 int *ip;
ip = (int *) malloc(100*sizeof(int));
```

- Some C compilers may require to cast the type of conversion.
- The **(int *)** means coercion to an integer pointer.
- Coercion to the correct pointer type is very important to ensure that the pointer arithmetic is performed correctly.
- It is good practice to use **sizeof()** even if you know the actual size you want as it makes code device independent (portable).
- **sizeof** can be used to find the size of any data type, variable or structure. Simply supply one of these as an argument to the function. So:

```
int i;
struct COORD
        {
                float x,y,z
        };
typedef struct COORD PT;
```

sizeof(int), sizeof(i),
sizeof(struct COORD) and
sizeof(PT) are all ACCEPTABLE

- In the above we can use the link between pointers and arrays to treat the reserved memory like an array. ***i.e*** we can do things like:

```
ip[0] = 100;
            or
for(i=0;i<100;++i)  scanf("%d",ip++);
```

- When you have finished using a portion of memory you should always **free()** it.
- This allows the memory ***freed*** to be available again, possibly for further **malloc()** calls
- The function **free()** takes a pointer as an argument and frees the memory to which the pointer refers.

Illegal indirection

- Consider:

```
*p = (char *) malloc(100);   /* request 100 bytes of memory */
*p = `y';
```

- There is mistake above as Malloc returns a pointer `and also p` does not point to any address.
- . The correct code should be:

```
p = (char *) malloc(100);
```

Calloc and Realloc

- There are two additional memory allocation functions, **`Calloc()`** and **`Realloc()`**.
- These are system defined functions whose prototypes are given below:

```
void *calloc(size_t num_elements, size_t element_size};
void *realloc( void *ptr, size_t new_size);
```

- `Malloc` does not initialize memory (to ***zero***) in any way.
- If you wish to initialize memory then use **`calloc`**.
- In Calloc, there is slightly more computationally expensive but, occasionally, more convenient than **malloc**.
- Also note that there is a different syntax between `calloc` and `malloc.`
- `The calloc` takes the number of desired elements, `num_elements`, and element_size, `element_size`, as two individual arguments.
- Thus to assign 100 integer elements that are all initially zero you would do:

```
        int *ip;
        ip = (int *) calloc(100, sizeof(int));
```

- Realloc is a function which attempts to change the size of a previous allocated block of memory.
- The new size can be larger or smaller. If the block is made larger then the old, contents remain unchanged and memory is added to the end of the block.
- If the size is made smaller then the remaining contents are unchanged.
- If the original block size cannot be resized then realloc will attempt to assign a new block of memory and will copy the old block contents.
- Note a new pointer (of different value) will consequently be returned. You **must** use this new value. If new memory cannot be reallocated then realloc returns NULL.
- Thus to change the size of memory allocated to the *ip pointer above to an array block of 50 integers instead of 100, simply do:

```
        ip = (int *) calloc( ip, 50);
```

Linked Lists

Let us see how linked lists are created in C. The linked list node structure is defined as follows:

```
typedef struct
        {       int value;
                ELEMENT *next;
        } ELEMENT;
```

- We can now try to grow the list dynamically:

```
        link = (ELEMENT *) malloc(sizeof(ELEMENT));
```

- This will allocate memory for a new link.

Basic I/O

- There are a couple of function that provide basic I/O facilities.

- The most common are: `getchar()` and `putchar()`. They are defined and used as follows:

```
int getchar(void) /*  reads a char from stdin */
int putchar(char ch)  /* writes a char to stdout, returns character
                          written. */

int ch;
ch = getchar();
```

Formatted I/O

Printf

- The function is defined as follows:

  ```
  int printf(char *format, arg list ...)
  ```

  /* it prints to `stdout`, the list of arguments according specified format string. Returns number of characters printed. */
- The **format string** has 2 types of object:
  - ***ordinary characters*** -- these are copied to output.
  - ***conversion specifications*** -- denoted by % and listed in the following Table

Table: Printf/scanf format characters

Format Spec (%)	Type	Result
c	char	single character
i,d	int	decimal number
o	int	octal number
x,X	int	hexadecimal number
		lower/uppercase notation
u	int	unsigned int
s	char *	print string
		terminated by \0
f	double/float	format -m.ddd...
e,E	"	Scientific Format
		-1.23e002

g,G	"	e or f whichever
		is most compact
%	-	print % character

scanf

- This function is defined as follows:
  ```
  int scanf(char *format, args....)
  ```
  /* reads from stdin and puts input in address of variables specified in `args` list. Returns number of chars read. */
- Format control string similar to `printf`
- The ADDRESS of variable or a pointer to one is required by `scanf`.
  ```
  scanf(``%d'',&i);
  ```
- We can just give the name of an array or string to scanf since this corresponds to the start address of the array/string.
  ```
  char string[80];
  scanf(``%s'',string);
  ```

Files

- Files are the most common form of a stream. The first thing we must do is ***open*** a file.
- The function `fopen()` does this: `FILE *fopen(char *name, char *mode)`
- `fopen` returns a pointer to a FILE. The `name` string is the name of the file on disc that we wish to access. The `mode` string controls our type of access. If a file cannot be accessed for any reason a `NULL` pointer is returned.
- Modes include:
  - ``r'' -- read,
  - ``w'' -- write and
  - ``a'' -- append.
- To open a file, we must have a stream (file pointer) that ***points*** to a FILE structure. So to open a file, called ***myfile.dat*** for reading we would do:
  ```
  FILE *stream, *fopen(); /* declare a stream and prototype fopen */
  stream = fopen(``myfile.dat'',``r'');
  ```
- It is good practice to check whether a file is opened correctly:

```
if ( (stream = fopen( ``myfile.dat", ``r")) == NULL)
        {       printf(``Can't open %s\n", ``myfile.dat");
                exit(1);
        }
......
```

Reading and writing files

- The functions fprintf and fscanf a commonly used to access files.

```
int fprintf(FILE *stream, char *format, args..)
int fscanf(FILE *stream, char *format, args..)
```

- These are similar to printf and scanf except that data is read from the ***stream*** that must have been opened with fopen().
- The stream pointer is automatically incremented with ALL file read/write functions. We **do not** have to worry about doing this.

```
char *string[80]
 FILE *stream, *fopen();
 if ( (stream = fopen(...)) != NULL)
        fscanf(stream,``%s", string);
```

- Other functions for files:

```
int getc(FILE *stream), int fgetc(FILE *stream)
int putc(char ch, FILE *s), int fputc(char ch, FILE *s)
```

- These are like getchar, putchar.
- getc is defined as preprocessor MACRO in stdio.h. fgetc is a C library function. Both achieve the same result!!

```
fflush(FILE *stream) -- flushes a stream.
fclose(FILE *stream) -- closes a stream.
```

- We can access predefined streams with fprintf ***etc.***

```
fprintf(stderr,``Cannot Compute!!\ n'');
fscanf(stdin,``%s'',string);
```

sprintf and sscanf

- These are like fprintf and fscanf except they read/write to a string.

```
int sprintf(char *string, char *format, args..)
int sscanf(char *string, char *format, args..)
```

- For Example:

```
float full_tank = 47.0; /* litres */
float miles = 300;
```

```
char miles_per_litre[80];
sprintf( miles_per_litre,``Miles per litre = %2.3f", miles/full_tank);
```