Second Year Computing Course:
Further procedural programming

Dr G. A. Wynn, Dr R. G. West, Prof. R. Willingale
Department of Physics & Astronomy, University of Leicester
September 2011

Introduction

In the first part of this course you learned the following:

- Editing, compiling, using Linux
- Writing simple programs using the 'C' language
- Variables, arithmetic expressions, input/output, loops (indefinite and definite), logical expressions, conditional execution, arrays

You might like to re-read the first year notes, to refresh your memory about some of these ideas. In this second part of the computing course you will build on this basic knowledge and introduce some more advanced concepts. A quick reference is given in the appendix to remind you of the contents of the 1st year course.

Getting started

The first thing to do is to access the Linux computer. In order to log to the machine you will need a user name and password for the SPECTRE and CFS service. You should have already received your CFS and SPECTRE account details in your First Year. If not, fill in the Computer Centre form and hand it in at the Library.

The X-Windows Client NX

You gain access to the SPECTRE machines using a PC running Windows XP (CFS). The program you must run on the Windows XP machine is an X-Terminal client called NX.

- *Log on to Windows XP (using your CFS username and password)*
- *Install NX (only needed once)*
    - o *All Programs→CFS Software 2→Select & Remove Software (click)*
    - o *NX Client 3.4.0-5 (click)*
    - o *Add (click)*
    - o *Done (click)*
- *Start up NX to log in to Ultra*
    - o *All Programs→CFS Software 2→NX Client 3.4.0-5→SPECTRE-UoL (click)*
- *Login to ULTRA using your SPECTRE username and password*
- *Start a Terminal Command Line window*
    - o *Applications->Accessories-> Terminal (click)*

You must make sure you type your username and password in the correct case (either capitals or lower case) as Linux operating systems are case-sensitive. Your password is known only to you and will not appear on the screen as you type. The first time you log on

you may enter an automatic process that forces you to change your password. Your new password must be at least 6 characters, 2 of which must be alphabetic and one of which must be numeric.

Dictionary words and pure numbers are not allowed. Use something that is not easy to guess, do not tell anyone what your password is and please don't forget what it is yourself.

When you start the Terminal Command Line you should get the system prompt which ends in a dollar sign

```
$
```

This is the operating system soliciting for commands from you. Anything you type and enter at this prompt will instruct the computer to do something.

If you make an error in typing a command, or want to repeat a previous command, you can use the command-line recall feature. Pressing CTRL-P (press and hold CTRL, press P, then release CTRL) will recall previous lines that you have typed. Pressing CTRL-N will advance to the next line. Using CTRL-B and CTRL-F you can move the cursor backward and forwards along the line to edit it.

Once you have finished using the Terminal Command Line Window you may logoff by typing exit (and hitting the 'RETURN' key)

```
exit
```

To log out of the Ultra session use

- System(click)→Log Out(click)

*** **Never leave a computer logged on to an account unattended!!** ****

The EMACS and NEDIT Editors

Information is stored on a computer in files. A file may contain many things, including plain text, a computer program, an image or a data set. In this workshop you will need to create and manipulate files. One way to create or edit a text file is to use software known as an editor. You can use **emacs** (which is more powerful) or **nedit** (which is simpler and more like using word). To invoke the editor you need to enter the commands

```
emacs filename &
```

or

```
nedit filename &
```

where `filename` is the name of the file you wish to edit. Invoke the emacs editor to create a new file by entering the command

```
emacs firstfile.txt &
```

If you have entered the command correctly a new window should open up on your terminal. The editor window shows the contents of the file `firstfile.txt`, which is empty at present. Type some text into the window and then save the contents of the file and exit emacs by holding down the control key while typing 'X' followed by 'C'. Answer any questions which appear in the bottom line of the emacs window. You have now created your first file. A listing of all of the files in the current directory (you will find out about directories in the next section) may be obtained by typing `ls`.

Entering this command should prompt the computer to list the name of your new file:

```
firstfile.txt
```

You may examine its contents using the command `more`. Entering the command

```
more firstfile.txt
```

into the computer should prompt it to reveal the text you typed into the emacs window.

In this workshop you will be asked to create files containing computer programs written in C. When you are required to do this give your file a sensible name followed by the extension .c, an example would be

```
emacs program1.c &
```

Some of the more useful emacs commands are listed below. You will have noticed that an ampersand (`&`) has been added at the end of the emacs command lines above. This runs emacs as a background job so that you can use the mouse to toggle between the emacs window and the original xterm window. When you have made an edit you can save the buffer using c-x c-s (see below) and then move to the xterm window WITHOUT killing emacs. Making edits, re-compiling and re-running your program is faster if you do this and you can still see your source code in the emacs window when you are running your program which can be useful.

EMACS Crib Sheet

Commands using "Control" ("C-" means hold down the Control, or CTRL, key at the same time as hitting the appropriate alpha or numeric key).

`C-x C-f`	Read a file (type the filename in the mini-window)
`C-x C-s`	Save the file in the specified filename (or type a name). If it's okay to save, just hit "Return"
`C-x k`	Remove a buffer from emacs. (Emacs can hold more than one file in memory at once, so you can edit multiple files simultaneously. Each file is held in a separate "buffer").
`C-x b`	Switch to a different buffer
`C-x C-c`	Kill emacs: if you have unsaved files, it will ask if you want to save them
`C-x 1`	If the emacs screen is split, revert to a single file display
`C-x 2`	Split the screen to display 2 files in emacs
`C-x i`	Insert the contents of another file into the current buffer
`C-g`	Kill the current command in the mini-window
`C-k`	Delete the current line
`C-d`	Delete the next character
`C-v`	Scroll down one screen length

Note : The mini-window is the single line at the bottom of the emacs window, which shows the emacs commands, and their results.

Files and directories

Once created, files can be stored in directories. It is good practice to store associated files together in directories. To make a directory called `Cworkshop` enter the command

```
mkdir Cworkshop
```

Now entering `ls` will prompt the computer to return

```
Cworkshop firstfile.txt
```

You can move the file firstfile.txt into the directory `Cworkshop` by entering the command

```
mv firstfile.txt Cworkshop
```

Entering `ls` should now only list the directory `Cworkshop`. You may list the contents of the directory Cworkshop by entering

```
ls Cworkshop
```

You can change your working directory by typing

```
cd Cworkshop
```

all of the files you now create will be stored in this directory. If you want to move up a level in the directory hierarchy, use

```
cd ..
```

If you want to find out the name of the current directory use

```
pwd
```

Files can be deleted using the `rm` command, eg.

```
rm firstfile.txt
```

You can make a copy of a file (creating a new file with a different name) using

```
cp firstfile.txt secondfile.txt
```

If you want to change the name of a file use the move command

```
mv firstfile.txt secondfile.txt
```

In this case the original file (firstfile.txt) will disappear.

As already mentioned above you can use CTRL-P, CTRL-N, CTRL-B and CTRL-F to edit previous command lines and save time and effort. CTRL-P will recover the previous line. Repeated use of CTRL-P will move back successive lines. If you overshoot use CTRL-N to go to the next line. CTRL-B moves the cursor to the left and CTRL-F to the right along the current line. Once the cursor is in the required position backspace will delete characters and typing will insert characters. When the line is as required use return to issue the command.

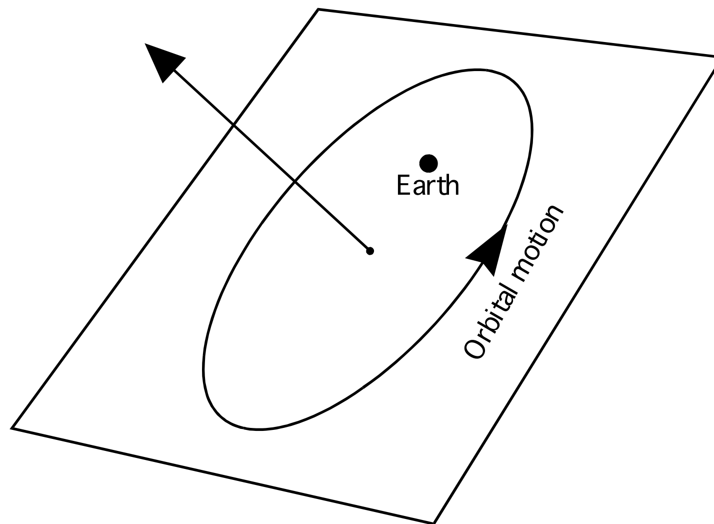
The problem: Tracking artificial satellites

The problem you will tackle in this course is to predict the positions of a set of artificial satellites in orbit about the Earth. Tens to hundreds of satellites are launched into orbit each year, and may orbit the Earth for decades before atmospheric drag slows them sufficiently for them to re-enter the atmosphere and burn up. In addition to the satellites themselves, the final stages of the launcher rockets will also continue to orbit for many months or years.

The orbits of artificial satellites and associated space junk are routinely tracked a US military organization known as NORAD. The orbital characteristics of non-military orbiting objects are released to the public on a regular basis. Using this information it is possible to predict the position of any catalogued satellite or piece of space debris at a time of interest. This is what you will be doing in this course.

Orbital mechanics

Once separated from its launch vehicle the primary force acting on an orbiting satellite is the gravitational pull of the Earth. If the Earth were a point mass the satellite would move in a planar elliptical orbit, with the Earth at one focus of the ellipse.



In this case predicting the position of an orbiting satellite or piece of space debris would be quite simple. Knowing the orientation of the orbital plane, the orbital period, and the position of the satellite at a fixed reference time would allow accurate predictions to be made years into the future.

The real world situation is rather more complicated. Firstly the Earth is not a point mass, and furthermore is not actually a perfect sphere, rather it is a flattened sphere (an oblate spheroid). Secondly the presence of the Sun, Moon and other planets will perturb the satellites motion from the elliptical ideal. Finally for satellites in orbits of less than ~1000km, the Earths atmosphere exerts a small but significant drag force. This drag force will depend on atmospheric density, which in turn depends on the strength of the Solar ultra-violet flux (which is time variable).

For this reason it is not possible to make accurate orbital predictions which will be valid for all time. It is necessary to make regular measurements of the satellites position, and use these measurements to update the model used to predict the future behaviour.

The NORAD data

The information released by NORAD comprises what are called *orbital elements*. The elements of an orbit are a set of eight numbers that characterise the size, shape and orientation of the elliptical orbital path that the satellite is following. You do not need to know the details of how orbital elements work, you will use existing subroutines which interpret the elements and return a satellite position at a time you request.

Program 1: Reading the orbital elements from a file

Copy the file `/home/z/zrw/under/satelements.txt` to your local directory, and inspect it by loading it into the emacs editor. ***BE CAREFUL NOT TO ALTER THE FILE!***

The file contains the orbital elements for 575 satellites. The orbit of each satellite is described by two lines in the file. The first two lines in the file describe the first satellite, the second two lines the second satellite, and so on. The entry for the first satellite looks like this:

```
AO-7
02224.67176888  1.0000e-04 101.7917 268.9850 0.0012289  82.8225 277.4242 12.53559109
```

The first line in the entry is the name of the satellite, and the second line contains eight numbers, which are the orbital elements. The numbers are as follows:

Epoch	A set of orbital elements represents a snapshot of the orbit at a particular point in time. The epoch specifies the time at which the snapshot was taken. In this file the first two digits of the epoch represent the year (02 = 2002), and then following digits the fractional day-within-year.
B* (Drag)	Although the Earths' atmosphere is very tenuous at orbital altitudes, nevertheless it exerts a force on a satellite that causes it to spiral downwards. The B* element describes the magnitude of the drag force on the satellite.
Inclination	The angle between the orbital plane and the equator. Units are degrees.
RA of the ascending node	This element also orients the orbit; it is the Right Ascension (RA) of the line of intersection of the orbital plane with the plane of the Earths' equator. Units are degrees.
Eccentricity	Eccentricity of the elliptical orbit (eccentricity e=0 is a circle).
Argument of perigee	The position angle around the elliptical orbit at which the satellite is closest to the Earth (perigee). Units are degrees.
Mean anomaly	The position of the satellite around the orbit at the epoch time. Units are degrees.
Mean motion	Orbital speed. Units are degrees per minute.

The following program will open the file and read the first entry:

```
ZZ
```

Exercise 1: *Type in this program, and save it in a file called* `readrec.c`*. Compile the program and run it. Alter the program so that it prints out the variables it has read, so that you can check that it is functioning correctly. You will need to:*

- *create a new file in which you can enter the text (*`emacs readrec.c`*)*
- *compile the program (*`gcc readrec.c –lm –o readrec`*)*
- *run the program (*`./readrec`*)*

You should be familiar with most of the program above from last years course. Re-read your First Year notes to refresh your memory if necessary. Note the use of ampersand as a prefix on the arguments in `fscanf (&epoch` etc.). This tells the computer to pass the address of the variable rather than the value of the variable to the function. The function (`fscanf`) is then able to write a new value to the variable.

Character strings

In last years' course you learned how to declare and use numeric variables, ie. variables whose values are numbers. There are many occasions when the values you will want to represent are not numeric. Say for example you were writing a program to manipulate a list of names. Obviously it is difficult to assign the value "*John Smith*" to an integer or to a real number! In this case you want a variable to hold the value of the name of the satellite.

Fortunately computer languages, C included, allow us to declare and use variables that represent characters and strings of characters (often called *character strings*, or *string variables*). In computer terminology a *character* is a single letter, number, or punctuation mark. For example '`A`', '`D`', '`h`', '`2`', '`*`' and '`)`' are all characters. A character string is simply one or more characters strung together in sequence.

The C language uses the data-type `char` to represent characters. A scalar variable declared to be of type `char` holds a single character. For example:

```
char cval;
cval = 'A';
```

In the C language a character string is represented as an array of characters (you will be familiar with arrays of numbers from last years course). In the program above a character string variable is defined which will hold the value of the satellite name:

```
char name[MAXSATNAMELEN];
```

This code declares a character string of length `MAXSATNAMELEN`. The satellite names are guaranteed to be 24 or fewer characters in length, so a symbolic constant is used to define the length of the strings that will hold these names.

We use the pre-processor directive `#define` to do this. You learned about the pre-processor (and #define) in the First Year course. In the program above we use `#define` like so:

```
#define MAXSATNAMELEN 24
```

Whenever the C pre-processor sees the character sequence `MAXSATNAMELEN` it will replace it with the sequence `24`. This means that the line of code which appears:

```
char name[MAXSATNAMELEN];
```

will be compiled as if it had been written

```
char name[24];
```

The standard library function `fgets()` is used read the satellite name from the data file. The function accepts three arguments: the string variable that will receive the result, the maximum number of characters to read (ie. the length of the string variable), and the handle to the open file from which the string will be read. It will read a complete line from the file (up to a newline character), but will only store up to the specified maximum number of characters in the string. It's important not to read more characters than the string can hold, otherwise your program will overwrite other variables and will probably crash. The function `fgets()` actually returns a value (the location of the string variable), but we don't need to use this so we don't assign this returned value to a variable, we just discard it.

If you want to print a string variable to the screen, you can do this using `printf()`, just like for numeric variables, but you should use the format specifier `%s` in place of the `%d` or `%f` which would be used for numeric variables.

Program 2: Structured data types

The record describing the orbit of each satellite comprises eight numbers (the orbital elements), plus an additional character string variable holding the satellite name. The entire file contains over 500 of these records. In principle it is possible to declare eight arrays to hold the orbital elements, plus an array of strings to hold the names, and to read the data from the file into these arrays.

This is not a very elegant solution however. Later on when you will use a pre-written subroutine to predict the orbit, and you will need to pass all of this information to that subroutine. Passing the eight elements (in the right order), plus additional arguments, would make that function very cumbersome to call.

There is a better solution. The C language allows you to define new *structured data types*, which act like a collection of related variables. Using this feature you can define a new data type to hold all of the information pertaining to a single satellite in a single variable. The following code does this:

```
struct satinfo {
  char name[MAXSATNAMELEN];
  int number;
  double epoch, bstar, incl, raasc;
  double ecc, argper, meananom, meanmot;
};
```

The `struct` keyword introduces the definition of the new data type, and is followed by a nametag that identifies the structured type within your program. This is followed by a block which declares each of the *members* of the structured type, describing their name, data type and in the case of array members, their size. Note that strings, integer and floating-point variables can be freely mixed within a structured data type.

Once this new type has been defined, you can declare variables of the new type, and use them:

```
struct satinfo {
  char name[MAXSATNAMELEN];
  int number;
  double epoch, bstar, incl, raasc;
  double ecc, argper, meananom, meanmot;
};

int main()
{
  /* Declare a new variable 'satp' */
  struct satinfo satp;

  /* Access the members */
  satp.number = 10;
  satp.epoch = 492.2345245;
  printf("number = %d, epoch = %f\n", satp.number, satp.epoch);
}
```

The dot operator ('`.`') is used when accessing the members of a structured variable, so satp.epoch refers to the `epoch` member of the structured variable `satp`.

Note that just like variables you cannot use a structured data type (ie. define variables using it) before it has itself been defined. Definitions of structured data types are generally placed near the start of any program (after the header files are `#include`'d, but before any executable code).

Exercise 2: *Add the definition of the structured type* `satinfo` *to the program* `readrec.c`*. Alter the program so that the first record is read into a variable of the new structured type. Check the results are what you expect by printing the values of the members to the screen.*

Program 3: User-defined functions

In the first part of the course you learned how to use one of a large number of *functions* in the C standard library to calculate mathematical functions. For example:

```
root_val = sqrt(val);
```

will calculate the square-root of the variable `val` and assign it to the variable `root_val`. Many useful mathematical functions are available as part of the standard C library (see the appendix). But what if you wish to calculate a mathematical function that isn't included in the standard C library?

The C language allows you to create *user-defined functions*, and to use them in the same way as the standard functions provided by the standard C library. User-defined functions can be used to implement virtually any mathematical function, or perform non-mathematical operations. User-defined functions can be as long and complicated as necessary, and are usually used to help to *structure* the code (to break the overall program into smaller, more understandable segments). In large C programs user-defined functions will form the bulk of the code.

In the NORAD data file some of the quantities are angles specified in degrees (they are: orbital inclination, RA of ascending node, argument of perigee, mean anomaly and mean motion). The orbit prediction function requires these to be supplied in radians (or radians per minute for mean motion), so your program will have to convert the angles from degrees to radians when they are read from the input file.

Converting from degrees to radians is straightforward (multiply by $\pi/180$), but may be performed often. As a simple example of what user-defined functions can do, we will define a new function to convert from degrees to radians.

Defining new functions

Creating user-defined functions is quite straightforward to do, but you need to supply the compiler with a few pieces of information about your new function.

Firstly you need to give the new function a name, in order to identify it. This name should be unique within your program. No other functions or variables that you define should have the same name, otherwise the compiler will not know which function you are referring to when you use that name. For this reason it is not a good idea to use the same name as a pre-existing function in the standard C library. Defining a new function called `sin()` for example could cause problems, as this function already exists in the standard library (it calculates the sine of an angle). For the degrees-to-radians function in this example, use the name `deg2rad`.

Secondly you need to be able to supply the angle that your function will convert. This value will be passed to the function as a *parameter* (alternatively called an *argument*). (You have seen parameters being passed to functions before – the `sqrt()` function in this program for example). Each parameter has a name and a type associated with it, for example your function will accept the velocity as a double-precision real (floating-point) number that we will call `v` inside the function.

Once your function has converted the angle, the value must be returned from the new function to be used in the expression or assignment where the function is called. For this reason you also need to specify what *type* of value the function will return. The function `deg2rad` will return a double-precision real (floating-point) number.

Lastly you need to write some code (the *function body*) that calculates and returns the value of the new function. In the case of `deg2rad` the body is just a few lines of code, but functions can be very complicated if required (ten, hundreds or even thousands of lines of code).

Bringing all these elements together, this is how the new `deg2rad` function would be defined:

```
#define PI 3.1415926535898

double deg2rad(double angle)
{
  double result = angle * PI/180.0;
  return result;
}
```

This code fragment defines a new function called `deg2rad`, which accepts a double-precision parameter called `angle`, performs some calculations and returns a double-precision result. All new function definitions are basically of this form:

```
result-type function-name(arg-type arg1, arg-type arg2, ...)
{
  <some code>
  return result;
}
```

The `return` statement is one you won't have come across, and its' purpose is to define the value that will be returned by the function. In `deg2rad` it returns the value that has been calculated for the angle in radians.

Once the new `deg2rad` function has been defined in your program you can use it in expressions in the usual way:

```
double incl;

incl = 90.0;            /* Define angle in degrees */
incl = deg2rad(incl);   /* Convert angle to radians */
```

Function prototypes

When you use variables in a C program you are used to obeying the rule that a variable must be *declared* before it can be used. For example in the following code fragment:

```
float pi;               /* Declare the variable */
pi = 3.1415279;         /* Define the variable */
```

the first line *declares* the variable and the second line *defines* the variable (ie. gives it a value). If these two lines were swapped around, the program would be invalid and the compiler would terminate with an error; *it is not legal to refer to a variable before it has been declared*. A similar rule must be obeyed when writing a new user-defined function: you should not refer to the function *before* it has been declared and/or defined.

A *function prototype* looks like just the first line of a function definition. The function body (the code which is executed when the function is invoked) is missing. The prototype of `deg2rad` would look like this:

```
double deg2rad(double angle);
```

A function prototype serves two purposes: firstly it declares the function (so when the compiler encounters the named function in an expression it can distinguish it from a variable), and secondly it provides the compiler with information about the parameters that the function accepts, and the type of the result it returns. This latter point is very important, because it allows the compiler to check that the parameters you are passing to a function when you call it are of the correct type (and perform a limited range of conversions if required). Proper use of function prototypes helps the compiler to ensure that you haven't made any typographical errors when writing your program.

Using function prototypes

The prototype of a user-defined function should be placed in a program *before* any code that refers to the function. The best place to put function prototypes in your program is very close to the top of the file, after any `#include` statements you are using, for example:

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

/* Function prototypes */
double deg2rad(double angle);

/* ... define your new functions here ... */

int main()
{
   /* ... main body of the program ... */
   /* ... use the function here ... */
}
```

Exercise 3: *Alter the program* `readrec.c` *so that the members* `incl`, `raasc`, `argper`, `meananom` *and* `meanmot` *are converted to radians once they have been read from the file. Compile and run the program, and check that the conversion is applied correctly by using your calculator.*

Prototypes of standard library functions

You may have been wondering the precise purpose of the `#include <stdio.h>` and `#include <math.h>` pre-processor directives which you have been using without explanation to this point. The major purpose of these directives is to include into your program the prototypes of the standard library functions that you wish to use. For example the standard header file `stdio.h` contains prototypes for input/output-related functions (eg. `printf`, `scanf`, `fprintf`, `fscanf`, etc.), and `math.h` contains prototypes for mathematical functions (`sin`, `cos`, `exp`, `sqrt`, etc.).

In all there are fourteen so-called *Standard headers* that declare prototypes for functions in the Standard library. For simple programs `stdio.h` and `math.h` will be all you need, though one or two more will be introduced in a later part of this course. See the Appendix for a summary of the Standard headers.

Program 4: Reading the whole data file

By now you should have a program that reads the first record of the database of orbital elements into a variable of a new structured data type, and converts five of the elements from degrees to radians, as required by the orbit prediction routine. This must be extended to read all of the entries in the data file.

You also need to define an array of the `struct satinfo` data type into which the data can be read. The code fragment below shows how to define such an array, and how to access the members of an element of such an array. Note, this fragment is an example of the sort of code required NOT a verbatim copy of what you must type. You are expected to adapt it to fit the particular environment of your program.

```
#define MAXSATS 1000

/* ... definition of 'struct satinfo' here ... */

int main()
{
   struct satinfo list[MAXSATS];
   int cnt;

   cnt = 27;

   /* Access the members of an element of an array of structured data
       types */
   list[cnt].epoch = 834.49587;
   list[cnt].number = cnt;
   list[cnt].meananom = deg2rad(list[cnt].meananom);
   ...
}
```

Last year you saw how to use an indefinite (`while`) loop to read an unknown number of records from an input file. Here is a reminder:

```
#include <stdio.h>

#define MAXSATS 1000

/* ... definition of 'struct satinfo' here ...*/

/* ... declaration of deg2rad() here ... */

int main()
{
  struct satinfo list[MAXSATS];
  int cnt;
  FILE *f;

  /* Open the file */
  f = fopen("satelements.txt", "r");

  /* Read records from the file until the end-of-file is reached */
  cnt = 0;
  while(!feof(f) && cnt<MAXSATS) {
      /* ... read record from file and convert angles ... */

      /* Remember position of the entry in the list */
      list[cnt].number = cnt;
          cnt++;
   }

   /* Close the file */
```

```
    fclose(f);
        printf("Read %d records from the satellite database\n", cnt);
}
```

Exercise 4: *Copy your program* `readrec.c` *to a new file called* `readfile.c`*. Use the Linux* `cp` *command to do this;*

```
cp readrec.c readfile.c
```

Alter `readfile.c` *so that it reads all the entries in the* `satelements.txt` *file and stores them in an array of type* `struct satinfo`*. Compile and run the program. How many records are there in the file?*

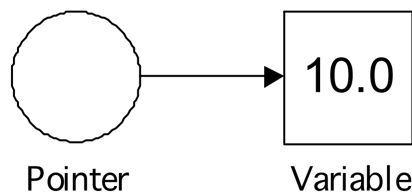
Program 5: Pointers

In this next section we will leave the subject of satellite orbital elements for a moment to introduce the concept of *pointers* in C. The example program you write here will be useful later on.

Pointers are an important and widely used part of the C language. They are an extremely powerful facility, and one of the prime reasons that 'C' is such a versatile and widely-used language.

You have already met and understood the concept of a *variable*. A variable can be thought of as a container that holds a value (the variables you have met so far have all held the values of numbers).

A pointer (or *pointer variable*) is different from a normal variable; rather than containing a value ***it points to a location that contains a value***.



Just like normal variables you must declare pointers before you use them. A statement that declares a pointer variable looks very similar to one declaring a normal variable:

```
int var;       /* Declare a variable */
int *ptr;      /* Declare a pointer */
```

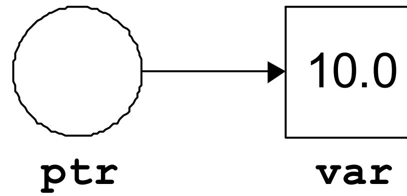
This example declares a variable `var` and a pointer `ptr`. Notice the presence of the '`*`' in the declaration of `ptr`; it is the '`*`' that identifies `ptr` as a pointer variable rather than a normal variable.

Assigning locations to pointer variables

The value of a pointer variable is *undefined* unless you actually assign something to it. You can make the pointer `ptr` point to the variable `var` like this:

```
ptr = &var;
```

The ampersand (`&`) operator returns the location of the variable it is applied to, so `&var` is the *location* of the variable `var`. You can assign this location to the pointer variable using the assignment operator in the usual way, using the assignment operator (=).



An important point to note here is that `ptr` is *not* a variable of type `int` (integer). It is a variable of type pointer-to-int. You *cannot* assign integer values to `ptr`, so for example the following two statements are *not* legal:

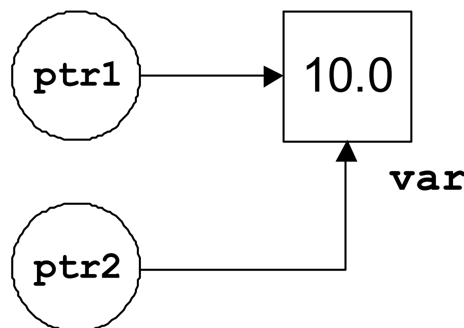
```
ptr = 10;
ptr = var;
```

The value of a pointer variable can be assigned to another pointer variable *of the same type*:

```
int val;
int *ptr1, *ptr2;

ptr1 = &val;            /* ptr1 points to 'val' */
ptr2 = ptr1;            /* ptr2 now points to 'val' */
```

In this example `ptr1` is initialised to point at the variable `val`, then the value of `ptr1` is assigned to `ptr2`. At this point both `ptr1` *and* `ptr2` will point at the variable `val`.



Pointers are not restricted to pointing at integer variables, they can point at variables of any type:

```
float fval, *fptr;
double dval, *dptr;
int ival, *iptr;
```

However you must be careful when assigning to pointers that they point to variables of the correct type:

```
iptr = &ival;           /* OK */
fptr = &fval;           /* OK */
dptr = &dval;           /* OK */

iptr = &fval;           /* Not legal, iptr is not pointer-to-float */
dptr = &fval;           /* Not legal, dptr is not pointer-to-float */

dptr = iptr;            /* Not legal, pointers are not the same type */
fptr = dptr;            /* Not legal, pointers are not the same type */
```

If you try to make an illegal pointer assignment the compiler will produce an error and won't compile your code.

De-referencing pointers

Once a pointer variable has been assigned a location of a variable (in other words once it points at a variable) you can read from and write to that variable by using the pointer. To do this you must place an asterisk ('*') in front of the name of the pointer, for example:

```
int val, other;
int *ptr;

ptr = &val;
*ptr = 10;              /* Assign value to variable 'val' */
other = *ptr;           /* Access value of variable 'val' through the
                           pointer */
printf("%d\n", other);  /* Will print '10' */
```

When referring to a pointer in this way you are accessing (reading from and writing to) the *location that the pointer points to, not the pointer variable itself*. This is called *de-referencing* the pointer.

Using pointers

Pointers have a number of uses, but primarily they are used when your program needs to know the location of a variable, rather than just it's value.

You have already seen how you can write a user-defined function to compute and return a single value:

```
/* Define new function to convert an angle from degrees to radians */
double deg2rad(double angle)
{
  double result = angle * PI/180.0;
  return result;
}
```

The `return` statement can only return the value of one variable. Suppose that you wish to write a function that returns two or more values. For example if you wanted to write a function that accepted Cartesian (x, y, z) coordinates and returned polar (θ, ϕ, r) coordinates, how would you do this? You can't use `return`, because this will only return a single value.

One way around this would be to write three distinct functions, to calculate each of θ, ϕ and r from (x, y, z), but this isn't very elegant. Ideally you would like to be able to write a single function that returns both values from a single call.

You might try writing this function as follows:

```
void cart2pol(double pos[3], double theta, double phi, double r)
{
  r = sqrt(pos[0]*pos[0] + pos[1]*pos[1] + pos[2]*pos[2]);
  theta = acos(pos[2]);
  phi = atan2(pos[1], pos[0]);
}
```

However this won't work as intended, for the following reason. Suppose you are using the function as follows:

```
void cart2pol(double pos[3], double theta, double phi, double r)
{
```

```
  r = sqrt(pos[0]*pos[0] + pos[1]*pos[1] + pos[2]*pos[2]);
  theta = acos(pos[2]);
  phi = atan2(pos[1], pos[0]);
}

int main()
{
  double pos[3], phival, thetaval, rval;

  pos[0] = 1.2;                 /* Assign a X coordinate */
  pos[1] = 2.3;                 /* Assign a Y coordinate */
  pos[2] = -1.5;                /* Assign a Z coordinate */

  /* Now call our function to calculate rval, thetaval */
  /* WON'T WORK AS INTENDED! */
  cart2pol(pos, thetaval, phival, rval);
}
```

The variables `phival`, `thetaval` and `rval` passed as input arguments to the function. Inside the function these arguments are referred to as the variables `r`, `phi` and `theta`, which are assigned the values of the polar coordinates calculated from the Cartesian coordinates passed in the array `pos`. The important point is that the variables inside the function are ***not the same variables*** as those used to call the function. The variable `r` inside the function is ***not the same variable*** as `rval`, and likewise `theta` is ***not the same variable*** as `thetaval`, and so on. So when the function assigns values to `theta`, `phi` and `r` the variables `thetaval`, `phival` and `rval` remain untouched.

When calling a function in 'C' you can use variables to supply any input parameters necessary, however when you do this you are passing the *value* of the variable, not the variable itself. This behaviour is called *pass-by-value*. This means that in the example above, when the function is called it is the values of `thetaval`, `phival` and `rval` that are passed, not the variables themselves. When the functions returns `thetaval`, `phival` and `rval` will have the same values they had before the function was called.

In order to allow the function `cart2pol()` to actually assign values to the variables `thetaval`, `phival` and `rval` you need to tell it the *location* of these variables, rather than their values. The way to do this is to pass pointers to the variables as parameters to the function, rather than the value of the variables. Using these pointers the function can be instructed where to write the values of , ϕ and r that are calculated.

Exercise 5: *Type in the following program and save it as the file* `cartpol.c`*. Compile and run the program. Check that the results it is producing are correct. The angles and ϕ will be printed in radians. (You will need to exit the program using CTRL-C).*

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

/* Convert from Cartesian to polar coordinates */
/* theta is in radians */
void cart2pol(double pos[3], double *theta, double *phi, double *r)
{
  *r = sqrt(pos[0]*pos[0] + pos[1]*pos[1] + pos[2]*pos[2]);
  *theta = acos(pos[2] / (*r) );
  *phi = atan2(pos[1], pos[0]);
}

/* This is the main body of the program */
```

```
int main()
{
  double pos[3], rval, phival, thetaval;

  /* Loop */
      while (1) {
          /* Prompt user to input Cartesian coordinates */
          printf("Enter X-coordinate: ");
          scanf("%lf", &pos[0]);
          printf("Enter Y-coordinate: ");
          scanf("%lf", &pos[1]);
          printf("Enter Z-coordinate: ");
          scanf("%lf", &pos[2]);

      /* Convert to polar coordinates and print out */
      cart2pol(pos, &thetaval, &phival, &rval);
      printf("Polar coordinates are r=%f, theta=%f, phi=%f\n",
                        rval, thetaval, phival);
  }
}
```

In this example program you can see that by passing a pointer to a variable rather than the value of the variable it is possible to write user-defined functions that calculate and return more than just a single value. The function calculates the results and writes them directly into the variables that our pointers point to. This is called *pass-by-reference*; you are passing a *reference* to a variable rather than its' value.

Because the function is using pointers to write directly to these variables, you no longer need the function to supply a return value. You will notice that both functions in the example program are defined to return a value of type `void`, and that they both lack a `return` statement. This is a way of telling the compiler that the function doesn't need to return a value by the `return` mechanism.

Another point to note is the use of the library function `scanf()` to read input from the keyboard. The arguments to this function are passed as pointers so that the function can assign the values it reads from the keyboard to the variables we want to contain them. This was introduced in the First Year course without any real explanation. It should now be clear why the ampersands (`&`) are needed on arguments to `scanf()`. The arguments are passed by reference, to allow `scanf` to write the values it reads into the variables that are to receive them. Also note that the descriptor used in **`scanf`** is **`%lf`** which denotes long float.

Program 6: Predicting satellite positions

Now you are ready to use the information read from the file to start predicting the orbits of satellites. To do this you will use a routine called `sgp4` that has been developed to calculate a position given a set of orbital elements and a time.

Copy the file `/home/z/zrw/under/sgp.c` to your local directory, and inspect it by loading it into the emacs editor. ***BE CAREFUL NOT TO ALTER THIS FILE!***

It contains quite a lot of code (over 300 lines) that defines four new functions. The function which you will be using is called `sgp4()` and starts about a quarter of the way down the file. You don't need to understand the working of this function, but you should take note of the first line of the function definition:

```
void sgp4(struct satinfo p, double tsince, double *pos)
```

The function takes the following arguments: variable of type `struct satinfo` (which will be one of the records your program has read from the input data file), a time (in minutes) after the reference time, and a pointer (`pos`) to an array which `sgp4()` will populate with the position of the satellite at the specified time.

Passing arrays to functions

An array in C is a variable with multiple values. These multiple values are arranged in successive locations in the computers memory. Specifying the location of the first element of the array, the number of elements in the array and the data-type of each element is sufficient to fully describe the array. This is the way that arrays are passed to functions in C.

A side effect of this is that all arrays in C are passed to functions by reference, not by value, and therefore a function that accepts an array as an argument can write to the array, to alter the values in the elements. The function `sgp4()` does this; after calculating the position and velocity of the satellite it can update the contents of the arrays passed as the `pos` and `vel` arguments.

Exercise 6: *Copy your program* `readfile.c` *to a new file called* `predpos.c`*. Insert the contents of the file* `sgp.c` *into this file at the correct place, which is in between the definition of the function* `deg2rad()` *and the definition of the main program body* `main()`*, after the definition of* `struct satinfo`*. The EMACS editor allows you to insert the contents of a file at the current cursor position, by using the keystroke sequence '*`C-x i`*' followed by the name of the file you wish to insert (see the EMACS crib sheet in the introduction to this course). Test that you have inserted the file correctly by compiling the program* `predpos.c`*.*

Calling sgp4()

Once the sgp4() function has been successfully incorporated into your new program `predpos.c`, you can call it to calculate satellite positions at different times. A call to `sgp4()` might look like this:

```
int i;
double tsince, pos[3];
struct satinfo list[MAXSATS];

/* ... read entries from database ... */

tsince = 120.0;
sgp4(list[0], tsince, &pos[0]);
```

This call would calculate the position of the satellite two hours (120 minutes) after the reference time, and return the position in `pos`.

The three elements of the `pos` array represent the X-, Y- and Z- coordinates of the satellite in a Cartesian coordinate system that is centred on the Earth, with a Z-axis aligned with North (the rotation axis of the Earth), and an X-axis which extends from the origin through the point (longitude=0, latitude=0) on the Earth surface. This coordinate system is fixed in the frame of the Earth; rotates as the Earth rotates.

Exercise 7: *Alter your program* `predpos.c` *to predict the position of the first satellite in your list over the first two hours after the reference time, at intervals of ten minutes. We suggest that you use an integer counter in a for loop to control the listing (*`for(i=0; i<np; i++)`*). In each pass of the loop increment the time (*`tsince=tsince+tinc`*). Convert the coordinates in the* `pos` *array from Earth radii to kilometres (1 Earth radius = 6378.135km), and print out the three coordinates for each sample. Check your results, which should be as follows:*

```
-7732.052486   1229.047851     -0.024441
-6538.702363   1584.927420   3985.300097
```

```
-3462.619698   1666.673075   6802.224666
  621.279734   1576.987209   7624.829934
 4537.123708   1378.869830   6212.608784
 7155.963132   1065.853340   2979.700209
 7730.660679    582.912586  -1125.989937
 6117.260405   -114.557196  -4903.732849
 2807.842974   -994.741536  -7256.681179
-1230.445417  -1910.791294  -7507.562963
-4843.924415  -2612.629233  -5584.049660
-7020.385353  -2814.537378  -2038.460833
```

Note that the last line in this table is at 110 mins because we started at zero.
Question: How long does the satellite take to complete 1 orbit?

Program 7: Plotting the satellite track

Because the coordinate system used by `sgp4()` to describe the satellite position is fixed with the rotating frame of the Earth, it is simple to convert from the Cartesian position vector returned by `sgp4()` into longitude and latitude you might be more familiar with.

Exercise 8: *Copy your program* `predpos.c` *to a new file* `trackplot.c`*. Add the function* `cart2pol()` *(which you wrote in Exercise 5) to* `trackplot.c`*, and alter the program so that it calculates the geographic longitude and latitude of the satellite from the Cartesian vector returned by* `sgp4()`*. Compile and test* `trackplot.c`*. Note that the angle θ returned by* `cart2pol()` *is zero at the North Pole and 180° at the South Pole. You will have to adjust this to match the more familiar definition of geographic latitude, where zero degrees lies at the Equator. Make sure that the longitude and latitude values you are calculating are in degrees.*

Once you can calculate geographic longitude and latitude, you can plot the track of the satellite in a familiar coordinate system, to more readily visualise the motion of the satellites. To do this directly from your program your best approach is to use one of a number of pre-written subroutine libraries that are freely available. For this course you will be using a package called PGPLOT.

A manual describing PGPLOT, and how to use them, can be borrowed from the Laboratory Preparation Room, or is available from the following Web page:

`http://www.star.le.ac.uk/~rw/compshop/`

Note that the PGPLOT library was originally designed to be called from programs written using the FORTRAN language, however it can quite easily be called from a C program.

Compiling and linking programs using the PGPLOT library

Before you get started with writing a program to use PGPLOT, the library needs to be made available to your program when you use the compiler/linker. The commands you need to use to do this are as follows:

The library of subroutines must be made available to your program when you use the compiler/linker and the commands you will need to produce a plot are as follows:

```
$ source /home/z/zrw/under/pgplot_c
```

```
$ cc  <name>.c  `cpgplot_link`  -o   <name>
$ <name>
```

The first line (starting with source) is only required *once* during a session. The `` `cpgplot_link` `` (use the single quotes at the top left of the keyboard) includes the necessary object libraries into the `cc` command. When you run the program the `cpgbeg()` function will prompt for a device name. The device type you should use on EXCEED is /XS. Hardcopy can be obtained using /PS (landscape) and /VPS (portrait). These create a PostScript text file called `pgplot.ps` which may be sent to the print queue using the following command:

```
$ lp -dA4 < pgplot.ps
```

You can get the hardcopy from the Print Station in the terminal room between the 1st and 2nd year Laboratories. You will need a Print Copycard, which you can get from the demonstrator.

Calling PGPLOT

The sequence of PGPLOT operations that you will need to perform is as follows:

1. Open a plotting device (for example a window on the screen, or a device to generate hard copy output).
2. Set up the range of X- and Y-coordinates your plot will cover.
3. Draw and annotate the axes
4. Plot your data points
5. Close the plotting device.

The PGPLOT function calls you will need are as follows:

```
/* Include the PGPLOT header file at the start of your program */
#include <cpgplot.h>
```

then in the main body of your program:

```
/* declare longitude and latitude variables */
float phi,theta

/* Open the plotting device, set up the environment. NOTE scanf should
not be used before cpgbeg in your program. There is a bug somewhere in
the code and cpgbeg will fail in this case.*/
cpgbeg(0, “?”, 1, 1);
cpgenv(180.0, -180.0, -90.0, 90.0, 1, 1);

/* Label the axes, including the name of the satellite plotted */
cpglab(“Longitude”, “Latitude”, list[0].name);

/* Loop to calculate positions every 15 seconds for a day */
for( <you fill this in> ) {
      ... calculate the satellite longitude/latitude

      /* Plot the position
      cpgpt1(phi, theta, -1);
}
```

```
/* Close the plotting device */
cpgend();
```

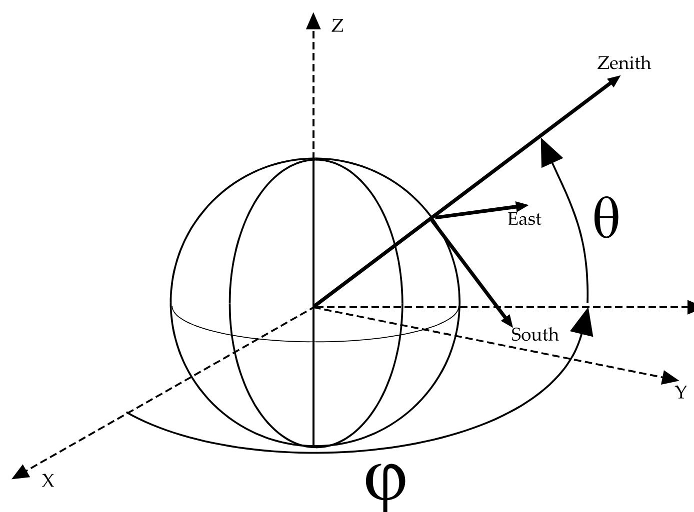
Exercise 9: *Alter your program* `trackplot.c` *to use PGPLOT to plot the position of a satellite in geographic longitude/latitude. Your program will be reading the orbital elements of 575 satellites from the NORAD data file. Add to your program the facility to choose which satellite is plotted by typing in a number corresponding to the index of a satellite in the list. Make a hardcopy of a few randomly chosen satellite tracks, and add them to your notes. Plot the track of satellite number 555, which is the International Space Station (ISS).*

Question*: Does the satellite cross the equator on successive orbits as expected?*

Program 8: Satellite tracks as seen from the ground

The tracks plotted in geographic coordinates do not give a very good idea of how a moving satellite might appear if viewed from a fixed spot on the Earth. A plot the track of a satellite across the sky as seen by an observer can be made by transforming the satellite position into a topocentric coordinate system (topocentric = centred on the observer).

The new coordinate system we choose to use is defined by the local zenith vector (which becomes the new Z-axis), and two vectors pointing directly East and South (which become the new X- and Y-axes respectively). The diagram below illustrates these new axes:



Once transformed into this Cartesian system, the coordinates are converted to a spherical polar representation of *azimuth* (angle eastwards along the horizon, from due North) and *elevation* (angle above the local horizon).

To perform this transformation firstly you need to know the coordinates of both the satellite and the observer in the same reference frame. The output from the function `sgp4()` is a Cartesian vector in a frame which rotates with the Earth. Given that a typical observer also rotates with the Earth, all you need to do is convert the geographic longitude/latitude of the observer into Cartesian form and you have what you need.

For an observer at longitude , latitude , the position of the observer is:

$\underline{\mathbf{P}}_{obs}$ = [cos cos , sin cos , sin]

The vector from the observer to the satellite is:

$\underline{\mathbf{r}} = [x_{sat} - x_{obs}, y_{sat} - y_{obs}, z_{sat} - z_{obs}]$

The coordinates of the satellite in the observers' reference frame are:

$$x' = -r_x \sin \quad + r_y \cos$$

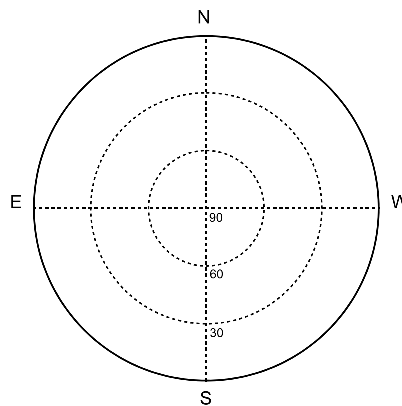
$$y' = r_x \sin \quad \cos \quad + r_y \sin \quad \sin \quad - r_z \cos$$

$$z' = r_x \cos \quad \cos \quad + r_y \cos \quad \sin \quad + r_z \sin$$

Once you have these new Cartesian coordinates you can use the function `cart2pol()` to convert them to azimuth and elevation (though remember you will have to manipulate the values of and which `cart2pol()` returns).

Whole sky chart

A whole sky chart represents the hemisphere of the sky as seen by the observer. The chart is circular, with the centre of the plot representing the point directly overhead the observer (elevation 90°), and the circular perimeter of the chart representing the horizon (elevation 0°). See the diagram below.



The following PGPLOT calls will set up the plotting environment and draw and annotate axes for a whole sky chart:

```
/* Set up the environment */
cpgenv(-90.0, 90.0, -90.0, 90.0, 1, -1);
  /* Draw axes */
cpgsfs(2);
cpgsls(2);
cpgsci(15);
cpgcirc(0.0, 0.0, 90.0);
cpgcirc(0.0, 0.0, 60.0);
cpgcirc(0.0, 0.0, 30.0);
cpgmove(-90.0, 0.0);
cpgdraw(90.0, 0.0);
cpgmove(0.0, -90.0);
cpgdraw(0.0, 90.0);
cpgsls(1);
cpgsci(1);
cpgtext(0.0, 90.0, "N");
cpgtext(0.0, -90.0, "S");
cpgtext(-90.0, 0.0, "E");
cpgtext(90.0, 0.0, "W");
cpgtext(0.0, 0.0, "90\\(2218)");
cpgtext(0.0, -30.0, "60\\(2218)");
cpgtext(0.0, -60.0, "30\\(2218)");
```

Exercise 10: *Copy your program* `plottrack.c` *to a new program* `plottrack2.c`. *Add the following features to the program:*

- *It should prompt the user for the longitude and latitude of an observer (in degrees)*
- *After calculating the satellite position it should transform the position into the local topocentric coordinate system of the observer and calculate the azimuth and elevation*
- *Implement both the spherical to Cartesian conversion, and the transformation to topocentric coordinates as user-defined functions*
- *Alter the plotting functions so that the positions are plotted on a whole sky chart (see above)*
- *Make hard copies of the sky tracks of a few randomly chosen satellites*
- *The program should print out the time and azimuth at which the satellite rises above and falls below the observers' local horizon (elevation = 0°) on each pass*

Question*: How long does the satellite chosen stay above the horizon in a single pass?*

Appendix: Quick C Reference

Skeleton program showing major features:

```
#include <stdio.h>               /* include standard headers */
#define MAXSATNAMELEN 24         /* macro definitions */

int main()                       /* declare main program function */
{                                          /* start scope of main */
     char name[MAXSATNAMELEN];   /* variable declarations for main */
     double epoch, bstar, incl;
     FILE *f;                              /* declare file pointer */
/* Open a file, read from file, close file */
     f = fopen("satelements.txt", "r");
     fgets(name, MAXSATNAMELEN, f);
 fscanf(f, "%lf %lf %lf\n",&epoch, &bstar, &incl);
  fclose(f);
}                                /* end scope of main */
```

Variables, types and declarations

The fundamental data types in C are:

`char`	a single character (usually 1 byte of 8 bits)
`short int`	usually 1 byte
`int`	an integer usually 2 bytes
`long int`	a large integer usually 4 bytes
`float`	single-precision floating-point usually 4 bytes
`double`	double-precision floating-point
`long double`	high-precision floating-point

The integer types including char can be qualified by unsigned or signed to determine whether or not a sign bit is included.

`unsigned int`	usually 2 bytes giving range 0 to 65535
`signed int`	usually 2 bytes giving range -32768 to 32767
`unsigned char`	1 byte giving range 0 to 255
`signed char`	1 byte giving range -128 to 127

Derived types are created using the declaration operators:

```
*             pointer, a prefix operator
&             reference, a prefix operator
[]            array, a postfix operator
()            function, a postfix operator
```

Constants

Integer constants are written as:

`9876`	assumed type `int`
`987654321L`	type `long`
`987654321l`	type `long`
`U9876`	type `unsigned int`
`u9876`	type `unsigned int`

Integers can be expressed in octal or hexadecimal:

`0123`	leading zero indicate octal constant
`0x134A`	leading `0x` (zero x) indicates hexadecimal

Floating point constants are written as:

`4.321`	type `double`
`4.3e-5`	type `double`
`3.1f`	type `float`
`3.1e-1F`	type `float`
`3.1l`	type `long double`
`3.1e-1L`	type `long double`

A character string constant is enclosed in double quotes.

```
"This is a character string constant"
```

Reserved identifiers

There is a set of identifiers reserved for use as keywords in C and C++ and these must not be used otherwise.

```
asm     continue   float    new        signed    try
auto    default    for      operator   sizeof    typedef
break   delete     friend   private    static    union
case    do         goto     protected  struct    unsigned
catch   double     if       public     switch    virtual
char    else       inline   register   template  void
class   enum       int      return     this      volatile
const   extern     long     short      throw     while
```

Input and output

```
scanf("%d",&value); // scan (read) standard input for value
printf("value typed %d \n",value); // write to standard output
fgets(line,sizeof(line),stdin); // get character string from file stream
sscanf(line,"%f",&ans); // scan string for value
nc=sprintf(textline,"an integer %d",ival); // write to a character string
```

The complete set of format specifiers is:

```
%d         integer decimal notation
%o         integer unsigned octal notation
%x         integer unsigned hexadecimal notation
%u         integer unsigned decimal
%c         single character
%s         string of characters
%e         floating point (single) exponential notation
%f         floating point (single) decimal notation
%lf        floating point (double) decimal notation
%g         floating point as %e or %f, whichever is shorter
```

The full list of escape characters is:

```
\n         newline
```

```
\t       horizontal tab
\v       vertical tab
\b       backspace
\r       carriage return
\f       form feed
\a       alert or bell
\\       backslash
\?       question mark
\'       single quote
\"       double quote
\0       null
\ooo     octal number
\xhhh    hexadecimal number
```

Operators

C has a very rich set of operators. Here is a list of common operators in order of precedence.

```
[]          subscripting                pointer[expr]
()          function call               expr(expr_list)
++          post/pre increment          lvalue++ or ++lvalue
~           complement                  ~expr
!           not                         !expr
-           unary minus                 -expr
+           unary plus                  +expr
&           address of                  &lvalue
*           indirection (dereference)   *expr
()          cast                        (type)expr
*           multiply                    expr*expr
/           divide                      expr/expr
%           modulo (remainder)          expr%expr
+           add (plus)                  expr+expr
-           subtract (minus)            expr-expr
<           less than                   expr<expr
<=          less than or equal          expr<=expr
>           greater than                expr>expr
>=          greater than or equal       expr>=expr
==          equal                       expr==expr
!=          not equal                   expr!=expr
&           bitwise AND                 expr&expr
^           bitwise exclusive OR        expr^expr
|           bitwise inclusice OR        expr|expr
<<          left shift bits             expr<<shift
>>          right shift bits            expr>>shift
&&          logical AND                 expr&&expr
||          logical inclusice OR        expr||expr
?:          conditional expression      expr?expr:expr
=           simple assignment           lvalue=expr
,           comma (sequencing)          expr,expr
```

In this table lvalue is an entity which can appear on the left hand side of an assignment, typically a variable name. This may be a simple variable, an array element or a pointer. You should be careful that an lvalue is what you intend, a pointer or a primitive type. The type of both sides of an assignment should be the same.

If you look at some C code you will often see composite operators like '+=' which means add and assign. These can be confusing and I suggest to begin with you avoid using these.

Compiler directives

```
#include <sys/pci.h>       // include a system header file
#include "pciadc.h"        // include a local header file
#define DEVICE_ID 0x0adc   // define a replacement macro
```

Commonly used macro definitions in header files:

```
EXIT_SUCCESS   // function integer return OK
EXIT_FAILURE   // funciton integer return not OK
```

Conditional statement blocks

The basic conditional statement block has the form:

```
if(expression1)
          statement1;
else if (expression2)
          statement2;
else
          statement;
```

If the statements require more than one line you must use curly braces to gather together the scope of each conditional:

```
if(expression1)
{
          statement1a;
          statement1b;
          ...
}
else if (expression2)
{
          statement2a;
          statement2b;
          ...
}
else
{
          statementa;
          statementb;
          ...
}
```

In either case the statement or statements following the (expression) are executed if the value of the expression is true or non-zero.

The `switch` statement can be used to select one of a number of alternative actions, depending on the value of a given expression. For example:

```
switch(expression) {
    case const1: statements;
         break;
    case const2: statements;
         break;
    ...
    default: statements
         break;
}
```

The expression is evaluated, and its' value compared with the constants `const1`, `const2`, etc., in turn. The statements following the matching constant are executed. If no constant following a `case` matches the value of the expression, the statements following `default` are executed instead.

Definite loops

A typical definite loop has the form:

```
int i, a[10];
for(i=0;i<10;i++)
{
        a[i]=i;
}
```

Indefinite and infinite loops

Indefinite loops come in two forms:

```
while(expression)
{
        body of loop
}

do
{
        body of loop
}
while(expression)
```

In the second variant the body of the loop is executed before the expression is evaluated thus ensuring the body is executed at least once.

An infinite loop can be set up using:

```
for(;;)
{
        ...
        if(finish loop expression)
                break;
        ...
}
```

Such a loop should be terminated using break as shown or return. The break statement can be used to terminate any for(), while() or do structure.

Functions and header files

The main program function: the integer `argc` is the number of command line arguments which are passed in character string array `argv`. Note `argv[0]` is the name of the program as it occurs on the command line:

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

The prototype declarations of all functions are held in header files. There are an enormous number of these files and an even larger number of prototype function declarations. The system wide header files are usually found at `/usr/include` on Unix and Unix-like systems. For example the floating-point mathematics functions are declared in `math.h`.

To use any of the functions you must declare them by including the appropriate header file at the top of your source file:

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

Standard header files

`assert.h`	Diagnostics and debugging
`ctype.h`	Character mapping and handling
`float.h`	Implementation-specific limits for values for floating-point numbers
`limits.h`	Implementation-specific limits for values which can be represented by integer numeric data types
`locale.h`	Localization (handling international character sets and cultural dependencies
`math.h`	Mathematical functions
`setjmp.h`	Non-local jumps
`signal.h`	Signal handling
`stdarg.h`	Facilities for writing functions with a variable number of arguments (parameters)
`stddef.h`	
`stdio.h`	Standard input & output functions
`stdlib.h`	Library of general utilities (eg. memory management, random number generators, string conversion functions, searching and sorting, communications with the execution environment)
`string.h`	Character string manipulation functions
`time.h`	Date and time functions

List of mathematical functions in C

Here is a list of some of the mathematical functions available in C. If you wish to use any of the following mathematical functions in your program, you will need to ensure that you have the line

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

in the first few lines of your program, and that you are using the `-lm` compiler switch when compiling your code.

`sin(x)`	Sine of x (x in radians)
`cos(x)`	Cosine of x (x in radians)
`tan(x)`	Tangent of x (x in radians)
`asin(x)`	Arcsine of x (result lies between $-\pi/2$ and $+\pi/2$)
`acos(x)`	Arccosine of x (result lies between 0 and $+\pi$)
`atan(x)`	Arctangent of x (result lies between $-\pi/2$ and $+\pi/2$)
`atan2(x, y)`	Arctangent of y/x (result lies between $-\pi$ and $+\pi$)

`exp(x)`	Exponential function
`log(x)`	Natural logarithm (base e) of x
`log10(x)`	Logarithm to base 10 of x
`powf(x, y)`	x to the power y (x^y)
`sqrt(x)`	Square-root of x
`fabs(x)`	Absolute value of x
`fmod(x)`	Returns the remainder of x/y, with the sign of x
`ceil(x)`	Returns the smallest integer not less than x
`floor(x)`	Returns the largest integer not greater than x

In the above table, 'x', and 'y' are of type `double`. All the above functions return `double` results.