A Brief History of C

C is a general-purpose language which has been closely associated with the UNIX operating system for which it was developed - since the system and most of the programs that run it are written in C.

Many of the important ideas of C stem from the language **BCPL**, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language **B**, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a DEC PDP-7. **BCPL** and **B** are "type less" languages whereas C provides a variety of data types.

In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

A Rough Guide to Programming Languages is available on-line for those of you that are interested.

Running C Programs

Objectives

Having read this section you should be able to:

1. Edit, link and run your C programs

This section is primarily aimed at the beginner who as no or little experience of using compiled languages. We cover the various stages of program development. The basic principles of this section will apply to what ever C compiler you choose to use, the stages are *nearly* always the same

The Edit-Compile-Link-Execute Process

Developing a program in a compiled language such as C requires at least four steps:

1. **editing** (or writing) the program
2. **compiling** it
3. **linking** it
4. **executing** it

We will now cover each step separately.

Editing

You write a computer program with words and symbols that are understandable to human beings. This is the *editing* part of the development cycle. You type the program directly into a window on the screen and save the resulting text as a separate file. This is often referred to as the *source file* (you can read it with the **TYPE** command in **DOS** or the **cat** command in **unix**). The custom is that the text of a C program is stored in a file with the extension .c for C programming language

Compiling

You cannot directly execute the source file. To run on any computer system, the source file must be translated into binary numbers understandable to the computer's Central Processing Unit (for example, the 80*87 microprocessor). This process produces an intermediate object file - with the extension `.obj`, the `.obj` stands for Object.

Linking

The first question that comes to most peoples minds is *Why is linking necessary?* The main reason is that many compiled languages come with library routines which can be added to your program. Theses routines are written by the manufacturer of the compiler to perform a variety of tasks, from input/output to complicated mathematical functions. In the case of C the standard input and output functions are contained in a library (`stdio.h`) so even the most basic program will require a library function. After linking the file extension is `.exe` which are executable files.

Executable files

Thus the text editor produces `.c` source files, which go to the compiler, which produces `.obj` object files, which go to the linker, which produces `.exe` executable file. You can then run `.exe` files as you can other applications, simply by typing their names at the **DOS** prompt or run using windows menu.

C's Character Set

C does not use, nor requires the use of, every character found on a modern computer keyboard. The only characters required by the C Programming Language are as follows:

- **A - Z**
- **a -z**

- **0 - 9**
- **space . , : ; ' $ "**
- **# % & ! _ {} [] () < > |**
- **+ - / * =**

The use of most of this set of characters will be discussed throughout the course.

Arithmetic Ordering

Whilst we are dealing with arithmetic we want to remind you about something that everyone learns at junior school but then we forget it. Consider the following calculation:

```
a=10.0 + 2.0 * 5.0 - 6.0 / 2.0
```

What is the answer? If you think its 27 go to the bottom of the class! Perhaps you got that answer by following each instruction as if it was being typed into a calculator. A computer doesn't work like that and it has its own set of rules when performing an arithmetic calculation. All mathematical operations form a hierarchy which is shown here. In the above calculation the multiplication and division parts will be evaluated first and then the addition and subtraction parts. This gives an answer of 17.

Note: *To avoid confusion use brackets*. The following are two different calculations:

```
a=10.0 + (2.0 * 5.0) - (6.0 / 2.0)
a=(10.0 + 2.0) * (5.0 - 6.0) / 2.0
```

You can freely mix **`int`**, **`float`** and **`double`** variables in expressions. In nearly all cases the lower precision values are converted to the highest precision values used in the expression. For example, the expression **`f*i`**, where **`f`** is a **`float`** and **`i`** is an **`int`**, is evaluated by converting the **`int`** to a **`float`** and then multiplying. The final result is, of course, a **`float`** but this may be assigned to another data type and the conversion will be made automatically. If you assign to a lower precision type then the value is truncated and not rounded. In other words, in nearly all cases you can ignore the problems of converting between types.

This is very reasonable but more surprising is the fact that the data type **`char`** can also be freely mixed with **`ints`**, **`floats`** and **`doubles`**. This will shock any programmer who has used another language, as it's another example of C getting us closer than is customary to the way the machine works. A character is represented as an ASCII or some other code in the range O to 255, and if you want you can use this integer code value in arithmetic. Another way of thinking about this is that a **`char`** variable is just a single-byte integer variable that can hold a number in the range O to 255, which can optionally be interpreted as a character. Notice, however, that C gives you access to memory in the smallest chunks your machine works with, i.e. one byte at a time, with no overheads.

The layout of C Programs

The general form of a C program is as follows (don't worry about what everything means at the moment - things will be explained later):

```
pre-processor directives
global declarations
main()
{
   local variables to function main ;
   statements associated with function main ;
}
f1()
{
   local variables to function 1 ;
   statements associated with function 1 ;
}
f2()
{
   local variables to function f2 ;
   statements associated with function 2 ;
}
```

There are five basic data types associated with variables:

- **int** - integer: a whole number.
- **float** - floating point value: ie a number with a fractional part.
- **double** - a double-precision floating point value.
- **char** - a single character.

void - valueless special purpose type which we will examine closely in later sections.

Having read this section you should have a clearer idea of *one* of C's:

1. input functions, called **`scanf`**
2. output functions, called **`printf`**

The % Format Specifiers

The `%` specifiers that you can use in ANSI C are:

	Usual variable type	Display
%c	char	single character
%d (%i)	int	signed integer
%e (%E)	float or double	exponential format
%f	float or double	signed decimal
%g (%G)	float or double	use %f or %e as required
%o	int	unsigned octal value
%p	pointer	address stored in pointer
%s	array of char	sequence of characters
%u	int	unsigned decimal
%x (%X)	int	unsigned hex value

Control Loops

Objectives

Having read this section you should have an idea about C's:

1. Conditional, or Logical, Expressions as used in program control
2. the **do while** loop
3. the **while** loop
4. the **for** loop

Looping the Loop

We have seen that any list of statements enclosed in curly brackets is treated as a single statement, a *compound statement*. So to repeat a list of statements all you have to do is put them inside a pair of curly brackets as in:

The `while` and `do while` Loops

You can repeat any statement using either the **`while`** loop:

`while(`*`condition`*`)` ***`compound statement;`***

or the **`do while`** loop:

`do` ***`compound statement`*** **`while(`*`condition`*`);`**

```
while (condition)
 {
   statementl;
   statement2;
   statement3;
 }
```

```
for ( counter=start_value; counter <= finish_value; ++counter )
  compound statement
```

which is entirely equivalent to:

```
counter=start;
while (couner <= finish)
 {
  statements;
  ++counter;
 }
```

Logical Expressions

So far we have assumed that the way to write the ***`conditions`*** used in loops and **`if`** statements is so obvious that we don't need to look more closely. In fact there are a number of deviations from what you might expect. To compare two values you can use the standard symbols:

> (greater than)
< (less than)
>= (for greater than or equal to)
<= (for less than or equal to)
== (to test for equality)

Select Paths with `switch`

While **`if`** is good for choosing between two alternatives, it quickly becomes cumbersome when several alternatives are needed. C's solution to this problem is the **`switch`** statement. The **`switch`** statement is C's multiple selection statement. It is used to select one of several alternative paths in program execution and works like this: A variable is successively tested against a list of integer or character constants. When a match is found, the statement sequence associated with the match is executed. The general form of the **`switch`** statement is:

```
switch(expression)
{
  case constant1:   statement sequence; break;
  case constant2:   statement sequence; break;
  case constant3:   statement sequence; break;
  .
  .
  .
  default:   statement sequence; break;
}
```

Functions and Local Variables

Now that the philosophy session is over we have to return to the details - because as it stands the **`demo`** function will not work. The problem is that the variable **`total`** isn't declared anywhere. A function is a complete program sub-unit in its own right and you can declare variables within it just as you can within the **`main`** program. If you look at the **`main`** program we have been using you will notice it is in fact a function that just happens to be called "main"! So to make **`demo`** work we have to add the declaration of the variable total:

```
demo()
 {
  int total;
  printf("Hello");
  total=total+1;
 }
```

Arrays

Objectives

Having read this section you should have a good understanding of the use of arrays in C.

Advanced Data Types

Programming in any language takes a quite significant leap forwards as soon as you learn about more *advanced data types* - *arrays* and *strings of characters*. In C there is also a third more general and even more powerful advanced data type - the **`pointer`** but more about that later. In this section we introduce the **`array`**, but the first question is, why bother?

There are times when we need to store a complete *list* of numbers or other data items. You could do this by creating as many individual variables as would be needed for the job, but this is a hard and tedious process. For example, suppose you want to read in five numbers and print them out in reverse order. You could do it the hard way as:

```
main()
{
 int al,a2,a3,a4,a5;
 scanf("%d %d %d %d %d",&a1,&a2,&a3,&a4,&a5);
 printf("%d %d %d %d %d",a5,a4,a3,a2,a1);
}
```

Structure and Nesting

Objectives

This section brings together the various looping mechanisms available to the C programmer with the program control constructs we met in the last section.

We also demonstrates a neat trick with random numbers.

It is one of the great discoveries of programming that you can write any program using just simple `while` loops and `if` statements. You don't need any other control statements at all. Of course it might be nice to include some other types of control statement to make life easy - for example, you don't need the `for` loop, but it is good to have! So as long as you understand the `if` and the `while` loop in one form or another you can write any program you want to.

If you think that a loop and an `if` statement are not much to build programs then you are missing an important point. It's not just the statements you have, but the way you can put them together. You can include an `if` statement within a loop, loops within loops are also OK, as are loops in `if`s, and `if`s in `if`s and so on. This putting one control statement inside another is called **nesting** and it is really what allows you to make a program as complicated as you like.

Our solution to the problem is as follows:

```
#include <stdio.h>

main()
 {
  int target;
  int guess;
  int again;

  printf("\n Do you want to guess a number 1 =Yes, 0=No ");
  scanf("%d",&again);

  while (again)
   {
    target = rand() % 100;
    guess  = target + l;

    while(target!=guess)
     {
      printf('\n What is your guess ? ");
      scanf("%d",&guess);

      if (target>guess) printf("Too low");
      else printf("Too high");
     }
```

```
      printf("\n Well done you got it! \n");
      printf("\nDo you want to guess a number 1=Yes, 0=No");
      scanf("%d".&again);
    }
 }
```

Functions and Prototypes

Objectives

Having read this section you should be able to:

1. program using correctly defined C functions
2. pass the value of local variables into your C functions

Functions - C's Building Blocks

Some programmers might consider it a bit early to introduce the C function - but we think you can't get to it soon enough. It isn't a difficult idea and it is incredibly useful. You could say that you only really start to find out what C programming is all about when you start using functions.

C functions are the equivalent of what in other languages would be called *subroutines* or *procedures*. If you are familiar with another language you also need to know that C only has functions, so don't spend time looking for the definition of subroutines or procedures - in C the function does everything!

A function is simply a chunk of C code (statements) that you have grouped together and given a name. The value of doing this is that you can use that "chunk" of code repeatedly simply by writing its name. For example, if you want to create a function that prints the word "**Hello**" on the screen and adds one to variable called **total** then the chunk of C code that you want to turn into a function is just:

```
printf("Hello");
total = total + l;
```

To turn it into a function you simply wrap the code in a pair of curly brackets to convert it into a single *compound statement* and write the name that you want to give it in front of the brackets:

```
demo()
 {
  printf("Hello");
  total = total + 1;
 }
```

Don't worry for now about the curved brackets after the function's name. Once you have defined your function you can use it within a program:

```
main()
 {
  demo();
 }
```

Functions and Local Variables

Now that the philosophy session is over we have to return to the details - because as it stands the **demo** function will not work. The problem is that the variable **total** isn't declared anywhere. A function is a complete program sub-unit in its own right and you can declare variables within it just as you can within the **main** program. If you look at the **main** program we have been using you will notice it is in fact a function that just happens to be called "main"! So to make **demo** work we have to add the declaration of the variable total:

```
demo()
 {
  int total;
  printf("Hello");
  total=total+1;
 }
```

Now this raises the question of where exactly **total** is a valid variable. You can certainly use **total** within the function that declares it - this much seems reasonable - but what about other functions and, in particular, what about the **main** program? The simple answer is that **total** is a variable that belongs to the **demo** function. It cannot be used in other functions, it doesn't even exist in other functions and it certainly has nothing to do with any variable of the same name that you declare within other functions.

This is what we hinted at when we said that functions were *isolated chunks of code*. Their isolation is such that variables declared within the function can only be used within that function. These variables are known as ***local variables*** and as their name suggests are local to the function they have been declared in. If you are used to a language where every variable is usable all the time this might seem silly and restrictive - but it isn't. It's what makes it possible to break a large program down into smaller and more manageable chunks.

The fact that **total** is only usable within the **demo** function is one thing - but notice we said that it only existed within this function, which is a more subtle point. The variables that a function declares are created when the function is started and destroyed when the function is finished. So if the intention is to use **total** to count the number of times the **>demo** function is used - forget it! Each time **demo** is used the variable **total** is created afresh, and at the end of the function the variable goes up in a puff of smoke along with its value. So no matter how many times you run **`demo total`** will only ever reach a value of 1, assuming that it's initialised to 0.

Constant Data Types

Constants refer to fixed values that may not be altered by the program. All the data types we have previously covered can be defined as **constant data types** if we so wish to do so. The **constant** data types must be defined before the main function. The format is as follows:

```
#define CONSTANTNAME value
```

for example:

```
#define SALESTAX 0.05
```

The **constant** name is normally written in capitals and does not have a semi-colon at the end. The use of **constants** is mainly for making your programs easier to be understood and modified by others and yourself in the future. An example program now follows:

```
#define SALESTAX 0.05
#include <stdio.h>
main()
 {
   float amount, taxes, total;
   printf("Enter the amount purchased : ");
   scanf("%f",&amount);
   taxes = SALESTAX*amount;
   printf("The sales tax is £%4.2f",taxes);
   printf("\n The total bill is £%5.2f",total);
 }
```

Strings

Objectives

This section brings together the use of two of C's fundamental data types, ponters and arrays, in the use of handling strings.

Having read this section you should be able to:

1. handle any string constant by storing it in an array.

Stringing Along

Now that we have mastered pointers and the relationship between arrays and pointers we can take a second look at *strings*. A *string* is just a *character array* with the convention that the end of the valid data is marked by a *null* '\0'. Now you should be able to see why you can read in a character string using `scanf("%s", name)` rather than `scanf("%s",&name)` - **name** is already a pointer variable. Manipulating strings is very much a matter of pointers and special string functions. For example, the **strlen(str)** function returns the number of characters in the string `str`. It does this simply by counting the number of characters up to the first null in the character array - so it is important that you are using a valid null-terminated string. Indeed this is important with all of the C string functions.

You might not think that you need a function to copy strings, but simple assignment between string variables doesn't work. For example:

```
char a[l0],b[10];
b = a;
```

does not appear to make a copy of the characters in **a**, but this is an illusion. What actually happens is that the pointer **b** is set to point to the same set of characters that **a** points to, i.e. a second copy of the string isn't created.

To do this you need **strcopy(a,b)** which really does make a copy of every character in **a** in the array **b** up to the first null character. In a similar fashion **strcat(a,b)** adds the characters in **b** to the end of the string stored in **a**. Finally there is the all-important **strcmp(a,b)** which compares the two strings character by character and returns true - that is 0 - if the results are equal.

Again notice that you can't compare strings using **a==b** because this just tests to see if the two pointers **a** and **b** are pointing to the same memory location. Of course if they are then the two strings are the same, but it is still possible for two strings to be the same even if they are stored at different locations.

You can see that you need to understand pointers to avoid making simple mistakes using strings. One last problem is how to initialise a character array to a string. You can't use:

```
a = "hello";
```

because **a** is a pointer and **"hello"** is a string constant. However, you can use:

```
strcopy(a,"hello")
```

because a string constant is passed in exactly the same way as a string variable, i.e. as a pointer. If you are worried where the string constant is stored, the answer is in a special area of memory along with all of the constants that the program uses. The main disadvantage of this method is that many compilers use an optimisation trick that results in only a single version of identical constants being stored. For example:

```
strcopy(b,"hello");
```

usually ends up with **b** pointing to the same string as **a**. In other words, this method isn't particularly safe!

A much better method is to use array initialisation. You can specify constants to be used to initialise any variable when it is declared. For example:

```
int a=10;
```

declares **a** to be an integer and initialises it to **10**. You can initialise an array using a similar notation. For example:

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

declares an integer array and initialises it so that **a[0]= 1**, **a[1] = 2** and so on. A character array can be initialised in the same way. For example:

```
char a[5]={'h','e','l','l','o'};
```

but a much better way is to write:

```
char a[6]="hello";
```

which also automatically stores a null character at the end of the string - hence **a[6]** and not **a[5]**. If you really want to be lazy you can use:

```
char a[] = "hello";
```

and let the compiler work out how many array elements are needed. Some compilers cannot cope with the idea of initialising a variable that doesn't exist for the entire life of the program. For those compilers to make initialisation work you need to add the keyword **static** to the front of the string declaration, therefore:

```
static char a[] = "hello";
```

Pointers to Structures

You can define a pointer to a structure in the same way as any pointer to any type. For example:

```
struct emprec *ptr
```

defines a pointer to an **emprec**. You can use a pointer to a **struct** in more or less the same way as any pointer but the use of qualified names makes it look slightly different For example:

```
(*ptr).age
```

is the age component of the **emprec** structure that **ptr** points at - i.e. an **int**. You need the brackets because '.' has a higher priority than '*'. The use of a pointer to a **struct** is so common, and the pointer notation so ugly, that there is an equivalent and more elegant way of writing the same thing. You can use:

```
prt->age
```

to mean the same thing as **(*ptr).age**. The notation gives a clearer idea of what is going on - **prt** points (i.e. **->**) to the structure and **.age** picks out which component of the structure we want. Interestingly until C++ became popular the **->** notation was relatively rare and given that many C text books hardly mentioned it this confused many experienced C programmers!

There are many reasons for using a pointer to a **struct** but one is to make two way communication possible within functions. For example, an alternative way of writing the complex number addition function is:

```
void comp add(struct comp *a , struct comp *b , struct comp *c)
 {
  c->real=a->real+b->real;
  c->imag=a->imag+b->imag;
 }
```

In this case **c** is now a pointer to a **comp struct** and the function would be used as:

```
add(&x,&y,&z);
```

Notice that in this case the address of each of the structures is passed rather than a complete copy of the structure - hence the saving in space. Also notice that the function can now change the values of **x**, **y** and **z** if it wants to. It's up to you to decide if this is a good thing or not!

Structures and Linked Lists

You may be wondering why **malloc** has been introduced right after the structure. The answer is that the dynamic allocation of memory and the **struct** go together a bit like the array and the **for** loop. The best way to explain how this all fits together is via a simple example. You can use **malloc** to create as many variables as you want as the program runs, but how do you keep track of them? For every new variable you create you also need an extra pointer to keep track of it. The solution to this otherwise tricky problem is to define a **struct** which has a pointer as one of its components. For example:

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

This defines a structure which contains a single **int** and - something that looks almost paradoxical - a pointer to the structure that is being defined. All you really need to know is that this is reasonable and it works. Now if you use **malloc** to create a new **struct** you also automatically get a new pointer to the **struct**. The final part of the solution is how to make use of the pointers. If you start off with a single 'starter' pointer to the **struct** you can create the first new **struct** using **malloc** as:

```
struct list *start;
start = (*struct list) malloc(sizeof(struct list))
```

After this start points to the first and only example of the `struct`. You can store data in the struct using statements like:

```
start->data=value;
```

The next step is to create a second example of the `struct`:

```
start = (*struct list) malloc(sizeof(list));
```

This does indeed give us a new `struct` but we have now lost the original because the pointer to it has been overwritten by the pointer to the new `struct`. To avoid losing the original the simplest solution is to use:

```
struct list *start,newitem;
newitem = (*struct list) malloc(sizeof(struct list));
start->prt=start;
start=newitem;
```

This stores the location of the new `struct` in ***newitem***. Then it stores the pointer to the existing `struct` into the **newitem's pointer** and sets the start of the list to be the `newitem`. Finally the start of the list is set to point at the new `struct`. This procedure is repeated each time a new structure is created with the result that a linked list of structures is created. The pointer start always points to the first `struct` in the list and the `prt` component of this `struct` points to the next and so on. You should be able to see how to write a program that examines or prints the data in each of the structures. For example:

```
thisptr=start;
while (1==1)
 {
  printf("%d",thisprt-> data);
  thisprt=thisprt->prt;
 }
```

This first sets `thisptr` to the start of the list, prints the data in the first element and then gets the pointer to the next `struct` in the list and so on. How does the program know it has reached the end of the list? At the moment it just keeps going into the deep and uncharted regions of your machine's memory! To stop it we have to mark the end of the list using a null pointer. Usually a pointer value of 0 is special in that it never occurs in a pointer pointing at a valid area of memory. You can use 0 to initialise a pointer so that you know it isn't pointing at anything real. So all we have to do is set the last pointer in the list to 0 and then test for it That is:

```
thisptr=start;
while (thisptr!=0)
 {
  printf("%d",thisprt->data);
  thisprt=thisprt-> prt;
 }
```

To be completely correct you should **TYPE cast 0** to be a pointer to the `struct` in question. That is:

```
while (thisptr!=(struct list*)0)
```

By generally mucking about with `pointers` stored in the list you can rearrange it, access it, sort it, delete items and do anything you want to. Notice that the structures in the list can be as complicated as you like and, subject to there being enough memory, you can create as many structures as you like.

You can use the same sort of technique to create even more complicated list structures. For example you can introduce another pointer into each structure and a pointer to the end of the list so that you can work your way along it in the other direction - **a doubly linked list**. You can create **stacks**, **queues**, **trees** and so on. The rest of the story is a matter of either inventing these data structures for yourself or looking them up in a suitable book.

File Handling

Objectives

So far we have entered information into our programs via the computer's keyboard. This is somewhat laborious if we have a lot of data to process. The solution is to combine all the input data into a file and let our C program read the information when it is required.

Having read this section you should be able to:

1. open a file for reading or writing
2. read/write the contents of a file
3. close the file

The Stream File

Although C does not have any built-in method of performing file I/O, the C standard library contains a very rich set of I/O functions providing an efficient, powerful and flexible approach. We will cover the ANSI file system but it must be mentioned that a second file system based upon the original UNIX system is also used but not covered on this course.

A very important concept in C is the **`stream`**. In C, the **`stream`** is a common, logical interface to the various devices that comprise the computer. In its most common form, a **`stream`** is a logical interface to a **`file`**. As C defines the term "file", it can refer to a disk file, the screen, the keyboard, a port, a file on tape, and so on. Although files differ in form and capabilities, all **`streams`** are the same. The **`stream`** provides a consistent interface and to the programmer one hardware device will look much like another.

A **`stream`** is linked to a file using an **`open operation`**. A **`stream`** is disassociated from a file using a **`close operation`**. The current location, also referred to as the current position, is the location in a file where the next file access will occur. There are two types of **`streams: text`** (used with ASCII characters some character translation takes place, may not be one-to-one correspondence between stream and what's in the file) and **`binary`** (used with any type of data, no character translation, one-to-one between stream and file).

To open a file and associate it with a **`stream`**, use **`fopen()`**. Its prototype is shown here:

```
FILE *fopen(char *fname,char *mode);
```

The `fopen()` function, like all the file-system functions, uses the header `stdio.h` . The name of the file to open is pointed to by ***fname*** (must be a valid name). The string pointed at for ***mode*** determines how the file may be accesed as shown:

Mode	**Meaning**
r	Open a text file for reading
w	Create a text file for writing
a	Append to a text file
rb	Open a binary file for reading
wb	Open a binary file for writing
ab	Append to a binary file
r+	Open a text file for read/write
w+	Create a text file for read/write
a+	Append or create a text file for read/write
r+b	Open a binary file for read/write
w+b	Create a binary file for read/write
a+b	Append a binary file for read/write

If the open operation is successful, `fopen()` returns a valid `file pointer`. The type **FILE** is defined in `stdio.h`. It is a structure that holds various kinds of information about the file, such as size.The `file pointer` will be used with all other functions that operate on the file and it must never be altered or the object it points to. If `fopen()` fails it returns a `NULL pointer` so this must always be checked for when opening a file. For example:

```
FILE *fp;

if ((fp = fopen("myfile", "r")) ==NULL){
  printf("Error opening file\n");
  exit(1);
}
```

To close a file, use `fclose()`, whose prototype is

```
int fclose(FILE *fp);
```

The `fclose()` function closes the file associated with *fp*, which must be a valid `file pointer` previously obtained using `fopen()`, and disassociates the `stream` from the file. The `fclose()` function returns 0 if successful and `EOF` (end of file) if an error occurs.

Once a file has been opened, depending upon its mode, you may read and/or write bytes to or from it using these two functions.

```
int fgetc(FILE *fp);
int fputc(int ch, FILE *fp);
```

The `getc()` function reads the next byte from the file and returns its as an integer and if error occurs returns `EOF`. The `getc()` function also returns `EOF` when the end of file is reached. Your routine can assign **fget()**'s return value to a **`char`** you don't have to assign it to an integer.

The **`fput()`** function writes the bytes contained in *ch* to the file associated with *fp* as an unsigned **`char`**. Although *ch* is defined as an **`int`**, you may call it using simply a **`char`**. The **`fput()`** function returns the character written if successful or **`EOF`** if an error occurs.

```
            /* 1 w.a.p. to reverse a number*/
#include<stdio.h>
void main()
{
      int a,d,rev=0;
      clrscr();
      printf("enter a number");
      scanf("%d",&a);
      while(a>0)
      {
            d=a%10;
            rev=rev*10+d;
            a=a/10;
      }
      printf("the reverse is %d",rev);
      getch();
}
output is:  enter a number768
            the reverse is 867
```

```
             /* 2 w.a.p. to print prime factor of a no*/
#include<stdio.h>
void main()
{
        int a,b;
        clrscr();
        b=2;
        printf("enter a no:");
        scanf("%d",&a);
        while(a>1)
        {
                if(a%b==0)
                {
                        printf("%d",b);
                        a=a/b;
                }
                else
                {
                        b++;
                }
        }
}
output is:
enter a no:36
 prime factor : 2233
```

```
         /* 3  w.a.p to check the no number is armstrong or not*/
#include<stdio.h>
void main()
{
        int a,sum=0;
        int n,arm;
        clrscr();
        printf("enter a no:");
```

```
      scanf("%d",&n);
      arm=n;
      while(n>0)
      {
            a=n%10;
            sum=sum+(a*a*a);
            n=n/10;
      }
      if(sum==arm)
      {
            printf("number is armstrong");
      }
      else
      {
            printf("number is not armstrong");
      }
      getch();
}
output is: enter a no:153
         number is armstrong.
```

```
            // 4 w.a.p. to print prime no between m & n.
#include<stdio.h>
void main()
{
      int m,n,b;
      clrscr();
      printf("enter uppercase value");
      scanf("%d",&n);
      printf("enter lowercase value");
      scanf("%d",&m);
      while(m<=n)
      {
            b=2;
            while(b<m)
            {
                  if(m%b==0)
                  {
                        break;
                  }
                  else
                  {
                        b++;
                  }
            }
            if(m==b)
            printf("%d",m);
            m++;
      }
      getch();
}
output is:enter uppercase value15
        enter lowercase value10
            11 13
```

```
/* 5 w.a.p to simulate a calculator*/
#include<stdio.h>
void main()
{     int a,b,c,i,j;
      char ch;
      do
      {
      clrscr();
      printf("1.add\n");
      printf("2.subtract\n");
      printf("3.multiply\n");
      printf("4.divide\n");
      printf("enter a choice\t");
      scanf("%d",&i);
      if(i==1)
      {
            printf("enter two no:");
            scanf("%d%d",&a,&b);
            c=a+b;
      }
      else if(i==2)
      {
            printf("enter two no:");
            scanf("%d%d",&a,&b);
            c=a-b;
      }
      else if(i==3)
      {
            printf("enter two no:");
            scanf("%d%d",&a,&b);
            c=a*b;
      }
      else if(i==4)
      {
            printf("enter two no:");
            scanf("%d%d",&a,&b);
            c=a/b;
      }
      else
      {
            printf("wrong choice");
      }
      printf("output is %d",c);
      fflush(stdin);
      printf("\nwant to perform any more(y/n)");
      scanf("%c",&ch);
      }while(ch!='n');
}
output is:  1.add
          2.subtract
          3.multiply
          4.divide
            enter a choice  1
            enter two no:12
            12
            output is 24
            want to perform any more(y/n)n.
              /* 6 w.a.p. to perform insertion sorting*/
#include<stdio.h>
void main()
{
      int a[5],i,j,k;
      clrscr();
      for(i=0; i<5; i++)
      {
            printf("enter the element of array");
            scanf("%d",&a[i]);
      }
      for(i=0; i<5; i++)
      {
            for(j=0; j<i; j++)
            {
                  if(a[i]<a[j])
                  {
                  k=a[i];
                  while(i!=j)
                  {
                        a[i]=a[i-1];
                        i--;
                  }
                  a[j]=k;
                  }
            }
      }
      for(i=0; i<5; i++)
      {
      printf("%d\n",a[i]);
      }
}
output is :enter the element of array12
         enter the element of array31
         enter the element of array58
         enter the element of array2
         enter the element of array1
                                       1
                                       2
                                       12
                                       31
                                       58
```

An Introduction to C++

Welcome to the inaugural edition of the ObjectiveViewPoint column! Here we will touch on many aspects of object-orientation. The word object has surfaced in more ways than you can count. There are OOPLs (Object-Oriented Programming Languages) and OODBs (Object-Oriented Databases), OOA (object-oriented analysis), and OOD (object-oriented design). We are sure you can come up with some OOisms of your own.

Our goal in this column is to explore object-orientation through practical object-oriented programming. This time, we look at C++, but in the future we will explore other areas of object-orientation. Learning an object-oriented language-a whole new way of programming-will pave the way for many exciting topics down the road.

Our intended audience consists of humble beginners to seasoned hackers. We assume that you have programmed in at least one procedural language, such as C or Pascal. Even if you are familiar with C++, please stay with us, you may learn some interesting new language features. Also, we will illustrate our points with many self-contained examples that you may later wish to incorporate into your own programs.

C++: A Historical Perspective

We begin our journey of C++ with a little history. C, the predecessor to C++, has become one of the most popular programming languages. Originally designed for systems programming, C enables programmers to write efficient code and provided close access to the machine. C compilers, found on practically every Unix system, are now available with most operating systems.

During the 1980s and into the 1990s, an explosive growth in object-oriented technology began with the introduction of the Smalltalk language. Object-Oriented Programming (OOP) began to replace the more traditional structured programming techniques. This explosion led to the development of languages which support programming with objects. Many new object-oriented programming languages appeared: Object-Pascal, Modula-2, Mesa, Cedar, Neon, Objective-C, LISP with the Common List Object System (CLOS), and, of course, C++. Although many of these languages appeared in the 1980s, many ideas of OOP were taken from Simula-67. Yes! OOP has been around since 1967.

C++ originated with Bjarne Stroustrop. In the simplest sense, if not the most accurate, we can consider it to be a better C. Although it is not an entirely new language, C++ represents a significant extension of C abilities. We might then consider C to be a subset of C++. C++ supports essentially every desirable behavior and most of the undesirable ones of its predecessor, but provides general language improvements as well as adding OOP capability. Note that using C++ does not imply that your are doing OOP. C++ does not force you to use its OOP features. You can simply create structured code that uses only C++'s non-OOP features.

A NEW FORM FOR COMMENTS.

It is always good practice to provide comments within your code so that it can be read and understood by others. In C, comments were placed between the tokens **/*** and ***/** like this:

```
/* This is a traditional C comment */
```

C++ supports traditional C comments and also provides an easier comment mechanism, which only requires an initial comment delimiter:

```
// This is a C++ comment
```

Everything after the **//** and to the end of the line is a comment.

THE CONST KEYWORD.

In C, constants are often specified in programs using **#define** . The **#define** is essentially a macro expansion facility, for example, with the definition:

```
#define PI 3.14159265358979323846
```

the preprocessor will substitute **3.14159265358979323846** wherever **PI** is encountered in the source file. C++ allows any variable to be declared a constant by adding the const keyword to the declaration. For the **PI** constant above, we would write:

```
const double PI = 3.14159265358979323846;
```

A **const** object may be initialized, but its value may never change. The fact that an object will never change allows the compiler to ensure that constant data is not modified and to generate more efficient code. Since each const element also has an associated type, the compiler can also do more explicit type checking.

A very powerful use of const is found when it is combined with pointers. By declaring a ``pointer to const'', the pointer cannot be used to change the pointed-to object. As an example, consider:

```
int i = 10;
 const int *pi = &i;
 *pi = 15;
// Not allowed! pi is a const pointer!
```

It is not possible to change the value of i through the pointer because ***pi** is constant. A pointer used in this way can be thought of as a read-only pointer; the pointer can be used to read the data to which it points, but the data cannot be changed via the pointer. Read-only pointers are often used by class member functions to return a pointer to private data stored within the class. The pointer allows the user to read, but not change, the private data.

Unfortunately, the user can still modify the data pointed at by the read-only pointer by using a type cast. This is called ``casting away the constness''. Using the above example, we can still change the value of i like this:

```
// Cast away the constness of the pi pointer and  modify i

*((int*) pi) = 15;
```

By returning a const pointer we are telling users to keep their hands off of internal data. The data can still be modified, but only with extra work (the type cast). So, in most cases users will realize they are not to modify that data, but can do so at their own risk.

There are two ways to add the const keyword to a pointer declaration. Above, when const comes before the ***** , what the pointer points to is constant. It is not possible to change the variable that is pointed to by the pointer. When when const comes after the *****, like this:

```
int i = 10;
int j = 11;
int* const ptr = &i;
// Pointer initialized to point to i
```

the pointer itself becomes constant. This means that the pointer cannont be changed to point to some other variable after it has been initialized. In the above example, the pointer ptr must always point at the variable **i**. So, statements such as:

```
ptr = &j;
// Not allowed, since the pointer is const!
```

are not allowed and are caught by the compiler. However, it is possible to modify the variable that the pointer points to:

```
*ptr = 15;
// This is ok, what is pointed at is not const
```

If we want to prevent modification of what the pointer points to and prevent the value of the pointer from being changed, we must provide a **const** on both sides of the ***** like this:

```
const int * const ptr = &i;
```

Remember that adding **const** to a declaration simply invokes extra compile time type checking; it does not cause the compiler to generate any extra code. Another advantage of using the **const** mechanism is that the C++ construct will be available to a symbolic debugger, while the preprocessing symbols generally are not.

DECLARATIONS AS STATEMENTS.

In a C++ program, a declaration can be placed wherever a statement can appear, which can be anywhere within a program block. Any initializations are done each time their declaration statement is executed. Suppose we are searching a linked list for a certain key:

```
int IsMember (const int key)
{
        int found = 0;
        if (NotEmpty())
        {
                List* ptr = head;
                // Declaration

                while (ptr && !found)
                {
                        int item = ptr->data;
                        // Declaration

                        ptr = ptr->next;

                        if (item == key)
                                found = 1;
                }
        }
        return found;
}
```

By putting declarations closer to where the variables are used, you write more legible code.

The Class: Data Encapsulation, Data Hiding, and Objects

Like a C structure, a C++ class is a data type. An object is simply an instantiation of a class. C++ classes have additional capabilities as the following example should show:

```
Vector v1(1,2),
Vector v2(2,3),
Vector vr;
vr = v1 + v2;
```

Vector is a class. **v1**, **v2**, and **vr** are objects of class **Vector**. **v1** and **v2** are given initial values through their constructor. **vr** is also initialized through its constructor to certain default values. The example illustrates a major power of C++. Namely, we can define functions on a class as well as data members. Here, we have an overloaded addition operator which makes our expression involving **Vector**s seem much more natural than the equivalent C code:

```
Vector v1, v2, vr;
add_vector( &vr , &v1, &v2 );
```

The ability to define these member functions allows us to have a constructor for **Vector**, code that creates an object of class Vector. The constructor ensures proper initialization of our **Vector**s.

Though not illustrated in the above example, a class can limit the use of its data members and member functions by non-member code. This is encapsulation. If class K defines member M as private, then only members of class K can use M. Defining M as public means any other class or function can use M.

Let's take a look at a trivial implementation of Vector that will show is a little about constructors, operators, and references.

```
#include <iostream.h>
class Vector
{
 public:
        Vector(double new_x=0.0,double new_y=0.0)       {
                if ((new_x<100.0) && (new_y<100.0))
                {
                        x=new_x;
                        y=new_y;
                }
                else
                {
                        x=0;
                        y=0;
                }
        }
        Vector operator +
                ( const Vector & v)
        {
                return
                (Vector (x + v.x, y + v.y));
        }
        void PrintOn (ostream& os)
        {
                os << "["
                        << x
                        << ", "
                        << y
                        << "]";
        }

private:
        double x, y;
};

int main()
{
        Vector v1, v2, v3(0.0,0.0);
        v1=Vector(1.1,2.2);
        v2=Vector(1.1,2.2);
        v3=v1+v2;
        cout << "v1 is ";
        v1.PrintOn (cout);
        cout << endl;
        cout << "v2 is ";
        v2.PrintOn (cout);
        cout << endl;
        cout << "v3 is ";
        v3.PrintOn (cout);
        cout << endl;
```

```
}
```

```
//PROGRAM OF C++ TO FIND NO OF VOWELS IN A GIVEN LINE OF TEXT
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
char line[80];
int no_vow=0;
cout<<”enter the line”<<endl;
cin.getline(line,80);
for(int I=0;line[I]!=’\0’;I++)
{
if(line[i]==’a’ || line==’e’ || line[i]==’i’ || line[i]==’o’ || line[i]==’u’
|| line[i]==’A’ || line==’E’ || line[i]==’I’ || line[i]==’O’ || line[i]==’U’)
no_vow++;
}
cout<<”number of vowels:”<<no_vow<<endl;
}
```

```
//PROGRAM TO ENTER THE 5 ELEMENTS OF 2 ARRAYS AND SUM OF ALL ELEMENT TO
THIRD ARRAY
#include<iostream.h>
#include<conio.h>
void main()
{
int arr1[5],arr2[5],arr3[5],i;
clrscr();
cout<<"Enter the 5 elements of array 1:";
for(i=5;i<5i++)
cin>>arr1[i];
cout<<"Enter the 5 elements of array 2:";
for(i=5;i<5i++)
cin>>arr2[i];
for(i=5;i<5i++)
{
arr3[i]= arr1[i]+ arr2[i];
cout<< arr3[i]<<endl;
}
getch();
}
```

```
//PROGRAM TO CALCULATE THE SALES SLIP
#include<iostream.h>
float tax(float);

void main()
{
float purchase,tax_amt,total;
cout<<”\n amount of purchase:”;
cin>>purchase;

tax_amt=purchase+tax_amt;
cout.precision(2);
cout<<”\n purchase is:”<<purchase;
cout<<”\n tax : “<<tax_amt;
cout<<”\n total:”<<total;
return 0;
}
float tax(float amount)
{
float rate=0.065;
return(amount*rate);
}
```

```
//PROGRAM TO GIVE DECLARATION FOR ARRAY OF 5 VOTERS
#include<iostream.h>
#include<conio.h>
struct voter
{
int id_no;
char name[30];
char address[50];
int age;
};
vote ward[20];

int main()
{
clrscr();
char ch;
for(int i=0;i<5;i++)
{
cout<<"\n"<<"vote no."<<i+1;
cout<<"\n"<<"Enter id_no.";
cin.get(ch);
cout<<"\n"<<"Enter name:";
cin.getline(ward[i].name,31);
cout<<"\n"<<"Enter Address";
cin.getline(ward[i].adress,51);
cout<<"\n"<<"Enter age";
cin>>ward[i].age;
}
for(i=0;i<5;i++)
{
if(ward[i].age>60)
{
cout<<"\n id:"<<ward[i].id_no;
cout<<\n name:"<<ward[i].name,31);
cout<<"\n");
}
}

}
```

```
// PROGRAM OF MERGE SORT
#include<iostream.h>
#include<conio.h>
void main()
{
int range[5], loc, lowest,i,j,n,t,x;
clrscr();
cout<<"Enter the array elements:\n";
cin>>n;
cout<<"Enter the array elements:\n";
        for(i=0;<n;i++)
          {
          if(lowest>range[j])
          {
                loc=j;
                lowest=range[j];
          }
          t=range[i];
          range[i]=range[loc];
          range[loc]=t;
          }
cout<<"\nsorted list is....\m";
for(i=0;i<n;i++)
cout<<range[i]<<endl;
}
```

```
//PROGRAM TO COUNT, CALCULATE AVERAGE, CALCULATE TOTAL
#include<iostream.h>
void main()
{
int number=1;
int total=0;
int count=0;
cout<<"\nEnter a number,0 to quit: \n";
while(number!=0)
cin>>number;
if(number==0)
cout<<"thank you, ending routline:\n";
else count++;
total+=number;
}
cout<<"total is "<<total<<"\n";
cout<<"count is "<<count<<"\n";
cout<<"the average is "<<total/count<<"\n";
 return 0;
}
```

```
//PROGRAM TO ACCEPT DATA AND CALCULATE SUM

#include<iostream.h>
#include<conio.h>
class sample
{
private:
        int a,b,c;
public:
        void getdata(int,int);
        void sum();
        void show();
};
coid samples::getdata(int,int)
{
a=i;
b=j;
}
void sample::sum()
{
c=a+b;
}
void sample::show()
{
cout<<"Sum="<<c;
}

void main()
{
int x,y;
clrscr();
sample s1,s2;
s1.getdata(10,20);
s1.sum();
s1.show();
cout<<endl;
cout<<"Enter two value:";
cin>>x>>y;
s2.getdata(x,y);
s2.sum();
s2.show();
getch();
}
```

```
//PROGRAM TO DELETE A RECORD FROM A FILE
#include<fstream.h>
#include<stdio.h>
#include<string.h>
class stu{
int rollno;
char name[25];
char class[4];
float marks;
char grade;
public:
        void getdata()
              {
                    cout<<"roll no:";     cin>>rollno;
                    cout<<"name:";        cin>>name;
                    cout<<"class:";       cin>>class;
                    cout<<"marks:";       cin>>marks;
                    if(marks>=75)         grade='A;
                    else if(marks>=60)  grade='B';
                    else if(marks>=50)  grade='C';
                    else if(marks>=40)  grade='D';
                    else grade='F';
                }
void putdata()
{
cout<<"rollno"<<rollno<<"\tname"<<name<<"\nmarks:"<<marks<<"\tgrade:"<<grade
<<endl;
}

int getno()          //accessor function
{
return rollno;
}
s1,stud;
void main()
ifstream fio("stu.dat","ios::in);
ofstream file("temp.dat",ios::out);
int rno;char found='f', confirm='n';
cout<<"enter roll no of student whose record is to be deleted \n";
cin>>rno;
while(!fi.eof())
{
fio.read((char*)&s1,sizeof(s1));
if(s1.getrno()<=stud.getno())
{
s1.putdata()
found='t';
cout<<"are your sure, you want to delete this record? (y/n)....";
cin>>confirm;
if(confirm=='n')
file.write(char*)&s1,sizeof(s1));
}
else
file.write(char*)&s1.sizeof(s1);
}
```

```
if(found=='f')
cout<<"record not found!!\n";
fio.close();
file.close();
remove("stud.dat");
rename("temp.dat",stu.dat");
fio.open(stu.dat",ios::in);
cout<<now the file contains \n";
while(!fio.eof()) break;
stud.putdata();
}
fio.close();
}
```