Functions

A *function* is a named group of statements which can be called (executed). A C++ program typically consists of a number of functions; execution begins with the *main* function.

WHY is it advantageous to write functions, rather than putting the entire program in the main function??

```
function main

 ----------
 ............ Task A
 ............
----------
 ----------
 ----------
............ Task A
 ............
----------
----------
-----------
............ Task A
 ............
 ----------
 -----------
```

Suppose your program must repeat the same task, say A, 3 times. It is much less error prone to write A only once and put it in a function which can be called.

Additionally, as A is now separate, it can be more easily debugged. You have used less storage, and A can be easily used in another program.

When you see repetition that is a hint that you should use a function.

```
 function main

  ------------------
  call A
----------
 ----------
call  A
----------
 ----------
call  A
---------
```

```
function A

   ----------
 ----------
 ----------
 ----------
```

```
function main
----------
 ----------
 ----------
&&&&&&
&&&&&&  Complex
&&&&&&    task
&&&&&&
 ----------
 ---------

■■■■■■■■■■■■■■■■■■■■■■■
```

When a program contains a complex task, it is best to delegate that task to a separate functions, where it can be more easily debugged and reused.

```
function main
   ------------------
   -------------------
   ------------------
   --------------------
```

```
function complex
 ------------------
---------------------
 ------------------
-------------------
```

Function Definition and Call

A function is defined as follows:

```
    return_type    name (formal parameter list){

            local variable declarations

            statements
    }
```

where:

return_type is the data type of the value that the function will return (if the function will not return a value, ‘void’ is used

name is the name by which the function will be called. it is chosen to reflect what the function does and must conform to the same rules as other identifiers in the program

formal parameter list is a list of declarations; one each value that the function will expect to be given (passed) when the function is called

local variable declarations are declarations for variable which are needed in the function . These variables are called local as they are created only when the function is called, and only exist for the duration of the function call. These variables do not exist outside the function.

statements are the C++ statements that perform the work of the function If the return type is not void, there should be a ‘return’ statement which indicates the value to be returned to the caller.

For example, the following function called sum_them, takes the two values that are passed to it when it is called, computes their sum, and returns it

```
int sum_them(int op1, int op2) {
    int  the_sum;

    the_sum = op1 + op2;
    return the_sum;
}
```

The function is then ‘called’ by using it’s name, and including the actual values to be communicated to the function within the parentheses. These values are called ‘arguments’ to the function. Arguments can be variables, as seen above, or values themselves, or expressions which result in the actual argument

For example a call to the function sum_them defined above might look like this:

```
 x = 14;
cin >> y;
result = sum_them(x,y);
```

Function Prototypes

C++ requires that a the compiler must see a function definition before it can process a call to that function.   When a function must be called before it is defined in a file, or the function that must be called is in another file, a prototype is used. A prototype is simply the ‘header’ line of the function definition , followed by a ‘;’.   For example:

```
 double foo  (double num);                 // prototypes for functions defined later
 int compute(int value1, int value2);

 int main () {
      ......
     ww = foo(count);
      .....
     answer = computer(a,b);
     ......
}

 double foo  (double num){
 // foo definition

}

int compute(int value1, int value2){
  //computer definition

}
```

Global Variables

Variables may be declared outside any functions. These are called ‘global’ variables, and may be shared by any function whose definition follows.   For example:

```
 double foo  (double num);                 // prototypes for functions defined later
 int compute(int value1, int value2);
 int value;

 int main () {
    int table;      // this variable is local to main, functions foo and compute do not know about it
     .....               // this function can use global variable value
  }

 double foo  (double num){
    double expon;    //only foo can use this value
                              // this function can use global variable value
}

int compute(int value1, int value2){
    double value;   // this local variable has the same name as the global variable value and ‘hides’ it from
       ........              //this function which would otherwise have access to it
  }
```

* The use of global variables can lead to problems, as many functions share the same data and an error in one function can then effect another. Global variables should ONLY be used when it is absolutely necessary (the information cannot be shared in any other way), or if efficiency is an issue (this may arise when a large array or other structure needs to be shared and parameter passing is time consuming).

Overloading Functions

C++ allows you to give two or more function definitions the same name. When this is done, that function is said to be *overloaded* .Why would you want to do that? Often a name is so appropriate to a task that it is clear to reuse the name that to make up another.

A classic example is MAX, a name which when given to a function indicates that the maximum of the arguments will be returned. Below the function MAX is 'overloaded'.

```
int main( ){
    int a,b,d,e,f;
    double q,r;
          .........
  answer = max(a,b);
  answer2 = max (d, e,f);

           ..........
  result = max(q,r);
}


int max(int v1, int v2, int v3){

}


int max(int a, int b){

}


double max(double x,  double y){

}
```

Definition for an overloaded function name must differ in their formal parameter list (called a signiture). This is necessary for the compiler to determine which function is actually being called. Note that the first two MAX functions above differ in the number of parameters, while the first and third differ in the data types of the parameters.