INTRODUCTION

In this week we cover the following topics

2.1 FUNCTIONS

- A function is a piece of code that is compiled separately. It can be given information in the form of parameters and return information as a return type. Its structure is as follows

```
return_type function_name(parameter list)
{
   many statements;
}
```

- Functions allow us to write modular code. This makes it easier to write large programs and to write code that can be used in many places. Objects are even better for this.

- The following function example illustrates

```
double gmean(double val1, double val2)
{
   return sqrt(val1 * val2);
}
```

An example of its use would be

```
void main()
{
   double x=3.6, y= 9.8, z;

   z = gmean(x, y);
}
```

In the example, `main` is said to *call* `gmean`.

2.1.1 Return Type

- The return type represents the type of information being sent back to the calling function.

- Its counterpart within the function is the `return` statement. It can be in the form of

```
return data_type;
```
or
```
return expression;
```

  where the expression evaluates to a value of the correct type.

- Whenever the function encounters the `return` statement it ceases operating and sends the appropriate value back to the calling function.

- Sometimes you don't want to return a value back. In this case the return type is set to `void`. E.g.

```
void foo(int x)
{
   if (x < 8) return;
   else    // do something
}
```

2.2.2 Parameters

- The parameter list is structured along the following lines

```
(type name1, type name2, ...)
```

- It is broken up into a list of variables, along with the type of the variable, separated by commas. The names of the variables must be unique within the function. E.g.

```
double gmean(double x, double y)
{
   return sqrt(x * y);
}
```

is OK but

```
double gmean(double x, double x)
{
   return sqrt(x * x);
}
```

is not OK;

2.2.3 Call By Value

- When a variable is sent to a function it is NOT the variable that is sent but its value. This is called *Call by Value*.

- The implication of this is that the variable in the parameter list has no connection to the variable in the calling function. Changes that happen to the variable in the called function are NOT reflected in the variable in the calling function.

- The following code illustrates

```
void foo(int x)
{
   x = 5;
    cout << "x in foo = " << x << "\n";
}

void main()
{
    int x = 3;

   foo(x);
    cout << "x in main = " << x << "\n";
}
```

will print the following

```
x in foo = 5
x in main = 3
```

2.2.4 Function Prototypes

- C++ compiles from top to bottom. A function being compiled will know about functions compiled above it but not below it. The following code illustrates.

```
int function_a(int x)
{
   /* some code */
}

void function_b()
{
   int y;

   y = function_a(y);
   y = function_c(y);
}

int function_c(int x)
{
   /* some code */
}
```

- The compiler will happily compile the first line in `function_b` since `function_a` will have already been compiled. However, it will generate a fatal error when it starts to compile the second line since it knows nothing about `function_c`.

- To solve this C++ uses function prototypes.

- Essentially, this involves creating a separate list of function declarations and putting them above all of your functions. E.g.

```
int function_a(int);
void function_b();
int function_c(int);

int function_a(int x)
{
   /* some code */
}

void function_b()
{
   int y;

   y = function_a(y);
   y = function_c(y);
}

int function_c(int x)
{
   /* some code */
}
```

- There are two things to note

  1. Notice the `;` at the end of each declaration. This informs the compiler that the line is a function prototype, not the actual function.
  2. Notice that we have only declared the types in the parameter list but not the names. You may include the names if you want but the compiler ignores them. All it is interested in is what type of variables are being passed.

- When the compiler compiles `function_b` it will have no problem with the second line since it knows there is a `function_c` that requires an `int` sent to it.

- Function prototypes have one other function. Namely strong type checking. Without function prototypes C++ was very lax about the types of variables sent to a function. E.g it would happily compile code such as this

```
void foo(int y)
{
   /* some code */
}

void main()
{
   double x = 3.0;

   foo(x);
}
```

- Note that `foo` expects to get an `int` but we sent it a `double`. If we had a function prototype placed above `foo` then the compiler would produce a warning about `foo` being called with the wrong type of variable.

- As programs get larger and more complex strong type checking helps reduce the instance of programming errors by enforcing some discipline on how we code.

2.2 CASTS

- Occasionally, we want to send a variable of one type to a function that expects a different type. To avoid

generating compiler errors C++ has the cast operator. An example of its use will help explain it.

```
void main()
{
  int x=4, y=7;

  double z = gmean((double) x, (double) y);
}
```

- Notice the `(double)` placed before `x` and `y` in the call to `gmean`. This is telling the compiler to explicitly cast the variable as `double`, rather than `int`.

- You cast a variable by placing the type you want it to be in `()`'s just before its use.

- Note that all the problems associated with turning floats into scalars, signed to unsigned, etc still exist. It is basically a message to the compiler that you really mean this.

2.3 SCOPE - GLOBAL Vs LOCAL VARIABLES

- In one of the earlier code examples we had an instance of declaring a variable within the function. E.g.

```
double gmean(double val1, double val2)
{
   double val3 = val1 * val2;

   if (val3 < 0) return -1.0;
   else return sqrt(val3);
}
```

- The variable `val3` only exists for as long as the function is being used. Once the function is finished, the memory set aside for `val3` is released back to the operating system.

- Beyond this, no other function can use `val3`. Even functions called by `gmean`. We say that the *scope* of `val3` is *local* to `gmean`. By scope, we mean the extent that a variable can be seen and accessed within the program.

- We can also have variables whose scope is *global*. Global variables can be seen by many functions and are created by declaring them outside of functions. The following illustrates

```
#include <iostream>

using namespace std;

int x = 6;

void foo();

void main()
{
   foo();
   cout << "x in main = " << x << "\n";
}

int y;
```

```
void foo()
{
   y = 3;
   x = 8;
   cout << "y in foo = " << y << "\n";
   cout << "x in foo = " << x << "\n";
}
```

will print the following

```
y in foo = 3
x in foo = 8
x in main = 8
```

A number of points need to made about this code

- As C++ compiles from top to bottom, global variables will only be visible to those functions declared after them. In the above example `main` and `foo` can see `x` but only `foo` can see `y`.

- Previously, we saw that variables sent as parameters to functions could not be changed by the called function. This was Call by Value. Here, global variables can be accessed by all the functions. Hence, using global variables gives us a way to have called functions change variables used within the calling function. However, this is bad programming practice. As programs grow larger and more complex it gives more scope for difficult to find errors creeping in. Some function may corrupt the contents of a global variable and it may be very hard to track down which one. As a general rule, the use of global variables should be avoided. What we need is a way of sending variables to functions that the called functions can change. C++

can do this but first, we must have an understanding of pointers.

2.4 POINTERS

- A pointer is a variable that contains the address in memory of another variable. They are declared like ordinary variables except they have an `*` in front of them to say they are a pointer. For example

```
int x, *y;
double *d;
```

- Here, `x` is a scalar that contains integer values but `y` is a variable that contains memory addresses of integer variables. `d` is a variable that contains memory addresses of double variables. It does not contain real values itself.

- It is VERY, VERY important not to confuse a pointer to a variable, with a variable.

- We can find the address of a variable by using the `&` address operator. For example

```
y = &x;
```

- In this example, we have used the `&` operator to put the memory address of `x` into `y` and we say *`y` is pointing to `x`*.

- We can use pointers to read and change the contents of the variables they point to. This is done by using the dereferencing operator `*`. For example

  ```
  *y = 5;
  ```

- Since we earlier set `y` to contain the address of `x`, setting `*y` to `5` sets `x` to `5`.

- Similarly we can have

  ```
  int z = *y;
  ```

  which will set `z` equal to what `x` is since `y` points to `x`.

- Note, `y = 5;` sets `y` pointing to memory location `5`. It does not change the contents of `x` and will almost certainly have disastrous consequences for your program. In other words, it will probably crash very quickly.

- Don't be confused with the `*` when declaring the pointers to its use when dereferencing a pointer.

- What happens if we hadn't set `y` pointing to `x`? Then, when we tried `*y = 5`, we would have been trying to set some unknown section of memory to `5`. Modern operating systems will prevent such operations. Usually you will get a *general protection fault* in Windows or a *segmentation fault* in UNIX as the program crashes. In fact, the majority of program crashes are usually the result of stray pointers. The

moral of the story is to be very careful with your use of them.

2.4.1 - Call By Reference

- Given this, we can now answer the question of how to get functions to change the contents of variables sent to them.

- Instead of sending a value to a function we can send an address. An example illustrates.

```
void foo(int *y)
{
   *y = 5;
   cout << “y in foo = " << *y << "\n”;
}

int main()
{
   int x = 3;

   foo(&x);
   cout << “x in main = " << x << "\n”;

   return 0;
}
```

will print the following

```
y in foo = 5
x in main = 5
```

- We can see that `foo` has been able to change the contents of `x` in `main`. This is because `foo` now has a

pointer variable `y` which contains the address location of `x` in `main`. By dereferencing `y` in `foo` we can access the contents of `x` in `main`.

- This is called *Call by Reference* and it is much more preferable to using global variables.

2.4.2 - Reference Operator

- In order to simplify Call by Reference in function calls, C++ uses the Reference Operator which makes use of the `&`. It works as follows

```
void foo(int &y)
{
   y = 5;
   cout << “y in foo = " << y << "\n”;
}

int main()
{
   int x = 3;

   foo(x);
   cout << “x in main = " << x << "\n”;

   return 0;
}
```

will print the following

```
y in foo = 5
x in main = 5
```

2.5 ARRAYS

- Arrays in C++ are a set of items, all of the same type (homogeneous).

- Each item within the array is called an *element*.

- Like all variables, arrays can be declared locally or globally.

- Arrays are specified by placing a set of square brackets after the variable and including a constant scalar value within the brackets. This value determines the number of elements in the array. Eg.

  ```
  double list[20];
  ```

  is an array called `list` containing 20 elements of type `double`.

- Arrays can also be declared in the following way

  ```
  int daysInMonth[] = {31, 28, 31, 30,
                       31, 30, 31, 31,
                       30, 31, 30, 31};
  ```

  which creates an array called `daysInMonth` containing 12 `int`'s and initialises each of them to the specified value.

- Whichever way you declare arrays, the size of them is fixed (static). You CANNOT declare arrays in the following manner

```
void test(int size)
{
  int array[size];  // ILLEGAL

  /* some code */
}
```

It is possible to dynamically create arrays on the fly which will be detailed later in the course.

- Each element in an array has a corresponding *index* (or subscript) value. The first element in the array has index 0, the second has 1, etc. E.g.

```
index -         0  1  2  3  4  5 .. 10 11
daysInMonth - 31 28 31 30 31 30 .. 30 31
```

It is VERY IMPORTANT to remember that the first element has index 0 and the last element has index size-1.

- You can use the index to access each element within the array. E.g.

```
int daysInJanuary = daysInMonth[0];
if (isleapYear(year))
    daysInMonth[1] = 29;
```

  - Doing things like

```
daysInMonth[-1]=30;
```

or

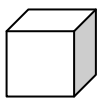
```
daysInMonth[12]=31;
```

will probably cause your program to catastrophically fail. Remember, indexes start at 0 and end at size-1. Unfortunately, there is no automatic checking on array indices. It is your responsibility to ensure you are using the correct index values. Do NOT try to access elements with index values outside this range. Accessing arrays outside their index range is a major cause of programs failing. Later on we will learn the use of more robust data structures with the Standard Template Library.

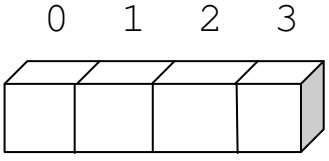
2.5.1 - Multi-Dimensional Arrays

- So far we have learnt about 1 dimensional arrays. C++ can have arrays of many dimensions. The most common instance is 2 dimensional arrays. Diagrammatically they can be visualised as

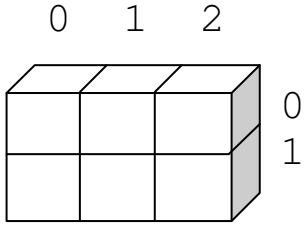
Single variable



1 Dimensional array



2 Dimensional array



- Examples of their syntax

```
int twoDee[2][3];
double threeDee[3][5][6];
```

For the `twoDee` think of it as 2 rows by 3 columns (see diagram above).
For the `threeDee` think of it as 3 planes by 5 rows by 6 columns.

- They can also be declared as follows

```
double twoDee[][3] = {{2.4, 5.3, 2.1},
                      {5.2, 9.0, 8.1}};
```

- Multi-dimensional arrays are used for many problems. E.g. a set of twenty samples with 5 measurements

```
int measures[20][5];
```

Or calculating the weather in a volume of space

```
double temp[1024][1024][16];
```

- Multi-dimensional arrays are accessed in similar ways to one dimensional. E.g.

```
measures[4][2] = 6;
temp[462][802][12] = 2.76;
```

2.5.2 - Arrays And Pointers

- Arrays and pointers work closely together.

- Arrays are implemented using pointers. The array is a pointer, which contains the address of the 1st element in the array. E.g.

```
int list[10], *listPtr;

listPtr = list;     // NOT listPtr = &list;
*listPtr = 2;       // same as list[0] = 2;
*(listPtr+3) = 6;   // same as list[3] = 6;
```

2.5.3 - Arrays As Function Arguments

- Arrays can be passed to functions as arguments but, as arrays are pointers, it will be Call by Reference. Because of this, changes made to an array within the called function will be reflected back in the calling function.

- The following code illustrates sending one and two dimensional arrays to functions.

```
void printOne(int list[], int size);
void printTwo(double twoDee[][3], int rows);
void printThree(int threeDee[][4][2],
                int planes);
```

```
void main()
{
   int list[20];
   double twoDee[2][3];
   int threeDee[3][4][2];

   printOne(list, 20);
   printTwo(twoDee, 2);
   printThree(threeDee, 3);
}
```

Note the use of the `[]`'s in the function prototypes. We could use pointers if we wanted since arrays are pointers. E.g.

```
void printOne(int *list, int size);
void printTwo(double *twoDee[3], int rows);
```

2.5.4 - Using Arrays In Functions

- Since an array is a pointer, when we send it to a function the function has no knowledge of how large it is. It only knows where the array starts.

  Hence, when we send an array to a function we also need to send its size. This is what we have done in the above prototypes.

- Given the size, we can use `for` loops to traverse that array. E.g.

```
void printTwo(double twoDee[][3], int rows)
{
   int row, col;

   cout.width(5);
   cout.precision(2);

   for (row=0; row<rows; row++)
   {
      for (col=0; col<3; col++)
         cout << twoDee[row][col];
      cout << "\n";
   }
}
```

2.6 DYNAMIC MEMORY ALLOCATION

2.6.1 - `new` and `delete`

- In C++, you often don't know how much memory you will need in a program when writing it. What is needed is a mechanism to allocate memory at run-time. C++ does this with the *`new`* and *`delete`* commands. E.g

```
int *ptr = new int;
delete ptr;
```

- In the first line, the `new` command tells the operating system to allocate enough memory for an integer type and then return the starting address of this memory. This address is put into `ptr`.

- Note that we now have an integer variable without a name. If we lose the address to this variable we will be unable to use it. This can cause serious memory

problems and it is very important to manage memory with extreme care in C++.

- In the second line the delete command is used. This is an instruction to the operating system that it can take back the memory that `ptr` points to. Once memory has been deleted it must never be used again. If you try to do this then you will very likely crash the program. Again, the lesson is to be very careful with memory management in C++.

2.6.2 – Dynamic Memory Allocation and Arrays

- Rarely do we allocate memory for one variable. What we usually do is allocate a lot of memory at once to create an array. It works in the following way.

```
double array[] = new double[size];
delete [] array;
```

- Note that the size of the array can be a variable. It doesn't have to be a constant. Once the memory is allocated we can use it in exactly the same way as an array.

- We could also use

```
double *array = new double[size];
```

- It is possible that the `new` command will fail. If that is so it will return a `NULL` value. It is always a wise thing to check for `NULL` before using the array. E.g.

```
if (array == NULL) exit(1);
```