Functions

In this lecture we will be primarily looking at controlling the flow of the program using loops and also at I/O (input/output).
Last week I mentioned the 3 methods of controlling the flow of a program:

- statements may only be obeyed under certain conditions (choice),
- statements may be obeyed repeatedly (loops),
- a group of remote statements may be obeyed (subroutines).

We looked at choice control, now we are looking at loop control or the ability to execute a series of statements repeatedly.

Scope
Before we go on to discuss functions we need to examine a few concepts, the first of these is scope.
So what is scope?

When we talk about scope we are generally referring to identifiers.

Scope = (of an identifier) is the range of the program statements within which the identifier is recognised as a valid name.

In computer programming, the scope of an identifier refers to where and when in the program the identifier can be referenced. Scope applies to all identifiers in a program, but it is mostly used for variables, functions and classes.

For example, the scope of a variable refers to where in the program the variable can be accessed and/or modified. For a function, to where the function can be called, and so on.
Scopes refer to the area of code where the identifier is 'visible'.

C++ Scope Rules:

- Every identifier must be declared and given a type before it is referenced (used)
- The scope of an identifier begins at its declaration
- If the declaration is within a compound statement, the scope of that identifier ends at the end of that compound statement. The identifier is referred to as local to that compound statement
- If the declaration is not within a compound statement, the identifiers scope ends at the end of the file. The identifier is referred to as global in scope.

Common scope levels:
Global scope: visible from the entire program
File scope: visible from a single source file – something we will look at when dealing with files

Local scope: visible only from the local function or block where the name was declared.

** Note: We place globals near the top of the program, above main but below any include or using statements.*

Example of Scope

```
#include <iostream>
using namespace std;
// constant - global in scope
const int DOG_YEARS = 7;

int main()
{     // variable - local in scope
      // can only be used with in main
      int userAge = 0;

      cin >> userAge;
      cout << "Your age in dog years is "
      << (userAge*DOG_YEARS) << endl;

      return 0;
}
```

Scope is usually hierarchical: a variable declared at local scope (for example, as a local variable inside a function) supersedes the same variable declared at global scope. All references to this variable inside the function will manipulate the locally-defined variable. Outside the function, they will manipulate the globally-defined variable.

Because of this automatic scope handling, careful management of variable, class and function names should be a requirement for any non-trivial program. It is considered good programming practice to make variables as narrow a scope as feasible so that different parts of your program do not accidentally interact with each other by modifying each other's variables.

Example:

```
static const int max_number_of_users = 100;

public class User {
    public static int number_of_users;

    public const char *name;

    public User (const char *new_name) {
      int n = number_of_users;
      n++;
      if (max_number_of_users < n)
```

```
          abort ();
      number_of_users = n;
      name = new_name;
    }
}
```

In this example,
`max_number_of_users` - global variable
`number_of_users` - class variable
`new_name` - local
`name` - instance variable

Binding

Binding = Determining which declaration of an identifier corresponds to a particular use (reference) of that identifier.

In computer science, binding refers to the creation of a simple reference to something which is larger and more complicated and used frequently. The simple reference can be used instead of having to repeat the larger thing.

Binding identifier references to declarations is the responsibility of the compiler. When a compiler finds a reference it searches for a matching declaration of the name. This search is conducted according to the following rules:

- A declaration of a constant or variable must match the identifier name exactly
- A declaration of a function must also match the number and parameter types
- The search proceeds upward, in the file, from the location of the reference
- If there is no matching declaration in the current code block (where the reference is found) and that block is contained in another code block the search continues into that enclosing block
- The search will not enter a code block contained in the code block being searched
- This search will continue, if necessary, until the global scope is reached and searched
- If not matching declaration is found the reference to the identifier is invalid and an “undeclared identifier” error message is displayed

A Word on Global Scope

Another contentious issue facing novice programmers is the use of identifiers that are global in scope.

Since a publication called “Use of Globals Considered Harmful” by Edsger Dijkstra way back in 1966 there has been something of a consensus amongst members of the software engineering community that identifiers, particularly those whose value can be change, should only be reluctantly declared as global.

When coding your assessments the use of global scope for constants, type definitions and function prototypes is allowed, the use of global variables will carry a penalty in assessments.

Functions and C++ Programs

A C++ program is a collection of one or more functions

- There must be a function called main( )
- Execution always begins with the first statement in function main( )
- Any other functions in your program are subprograms and are not executed until they are called

The Oddity of `main()`

Before we look at those in more detail a quick word about main, every C++ program has at least one function: `main()`. When the program is started `main()` is called automatically. `main()` is effectively the starting point for execution of the program, it is called by the operating system. It is strange in that it is part built in and part defined. You write exactly what `main()`does, but is always returns an integer (or should) and is defined as the effective start of the program.

Functions

What is a Function?

A function is a mechanism for encapsulating a section of code and referencing it by use of a descriptive identifier.

A function is a block of statements that are written to perform a specific task. Functions are in effect sub-programs that can act on data (doesn’t mean they have to) and return a value.

About Functions
Functions provide a mechanism for code reuse by enabling you to group code that would otherwise be replicated throughout your program in a single area.

They are a fundamental unit of program decomposition (or modularity), programs can be developed separately that are then combined to produce a finished program consisting of a group of cooperating functions.

With the exception of `main()` every C++ functions are called or invoked from other functions in your code.

The calling function and the called function may communicate, through information exchange, in a number of ways.

The use of functions makes testing a program easier since each function can be tested separately from the rest of the program.

Each function has a name and when that name is encountered the execution of the program braches off to the first statement of that function and continues until a return statement or the end of that function. When the functions returns execution continues on the next line of the calling function.

Function Groups
There are two main function groups in C++, built in functions and user-defined functions.

The types of functions in C++ can be grouped in the following two types:

- **Built-In Functions** are those that are included with your C++ compiler and can be called from your program. e.g. sqrt(), sin(), toupper(), etc. These definitions of these functions are defined in the C++ standard library and including the relevant header file includes the function definitions.

- **User-Defined Functions** are blocks of statements written to perform a specific task by the programmer. They are called "user defined" functions because they are defined and coded by the user of the language / compiler. i.e. the programmer.

All user-defined functions must be declared, activated (called) and defined.
A user-defined function is declared and given a name by the programmer; it is called or invoked by using its name (see how functions are called). Remember when naming functions the conventions on naming and how to comment your code.

There are many built in functions contained within the standard library and as such a general rule is when you need your program to do anything you should see if there is a built-in function (or combination of built-in functions) that do the required job. If suitable functions are not available then you should write your own user-defined function.
With games you'll be writing most of your own functions ☺

We will be mainly concerned with writing our own functions.

Function Definition
A function is invoked to carry out a particular task

Every function must have an implementation or definition that contains the statements that will be executed when the function is invoked.

The function definition is a body (block of code) that may contain any valid C++ statements.

Definitions can be in any order, it is considered good practise to declare `main()` first

Function definitions may not be nested

Basic Syntax
<return type> ***<function name>*** *(<type> <parameter name>, ...)*
{
// Definition statements – body of function
<return statement>
}

<return type>
The data type the function returns

<function name>
The name of the function, chosen by the coder

<type>
The data type of each parameter

<return statement>
The data value to return, may be omitted if <return type> is null

Examples
```
int GetAge()
```

For this function:
Return type = integer
Function name = GetAge
There are no parameters

```
string GetName()
```

For this function:
Return type = string
Function name = GetName
There are no parameters

```
void UserPrompt(string prompt)
```

For this function:
There is no return type (reserved word void)
Function name = UserPrompt
There is one parameter
The parameter type is string
The parameter name is prompt

```
int AddTwoNumbers(int number1, int number2)
```

For this function:
Return type = integer
Function name = AddTwoNumbers
There are two parameters (notice the separation of the parameters by a comma)
Parameter type for both is integer
The parameter name is Number1 and Number2 respectively

Example Function

```
int AddTwoNumbers(int number1, int number2)
{
      int total = number1 + number2;
      return total;
}
```

The `return` statement

We have seen the return statement in the function main and in function above.

The return statement should return a type that matches the return type declared at the beginning of the function definition.

The return statement accomplishes the following things:

- Immediately terminates execution of the function within which the return statement occurs

- Replaces the function invocation with the value of the variable or expression specified in the return statement
- Resume execution of the calling function

A function definition may contain more than one return statement, however only one will be executed on a given call to that function.

Void functions do not need a return statement. You can add them to make the program return when you want rather than returning when the end of the function (closing '}' bracket) is reached.

```
int userAge = GetAge();
```

In this example the value of userAge is set to the value that is returned from GetAge().

Function Communication and Scope
With the return statement we can pass information, in the form of a single value, back to the calling function.
In many cases it is also necessary to pass information from the calling function to the called function.

One of these methods is identifiers with global scope.

Consider the following program:

```
double CircleCircumference(double radius)
{
	return (PI * (radius*2));
}
```

Where should PI be declared and set?
We could declare PI as a global constant, as its value doesn't change, it would then be accessible to the function.

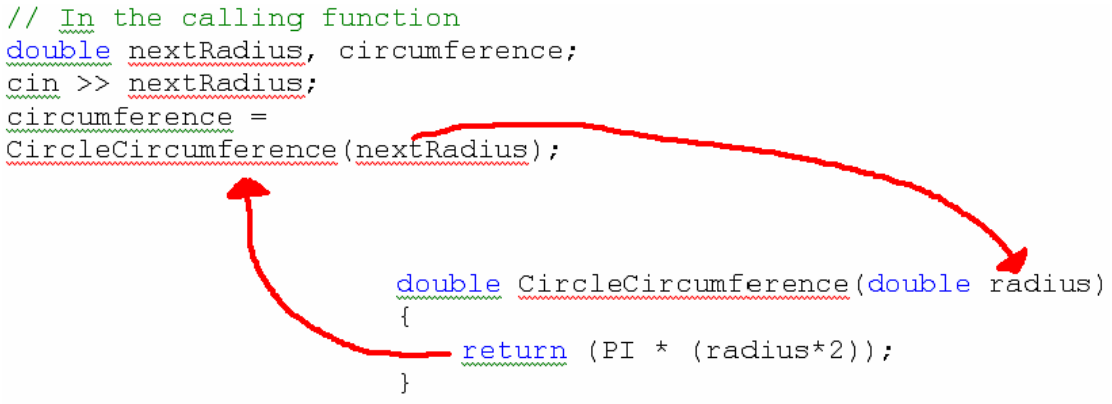
Don't forget that in general we should avoid globals, particularly global variables.

Function Communication and Parameters
The calling function can pass information to the called function by means of parameters.

Consider the calling of this function:

```
double CircleCircumference(double radius)
{
	return (PI * (radius*2));
```

```
}
```



The value of nextRadius in the calling function is copied to the variable radius in the called function.
The `CircleCircumference` function can then make use of that to perform its operations.

The Formal Parameter List
The function definition must provide a list of declarations of variables that are used for communication.
These variables are declared within the parentheses following the function name.

<return type> ***<function name>*** *(<type> <parameter name>, ...)*
{
// statements – body of function
<return statement>
}

<type> and <parameter name> combined are known as a formal parameter.
Multiple types and names are known as formal parameters.

A formal parameter is in effect a placeholder into which a calling function will pass a value.

The scope of the formal parameter is the body of the function definition.

Formal parameters may be of any valid C++ type.

Formal parameter declarations are comma separated.

A function may have no formal parameters or as many as needed.

The formal parameter list coupled with the return type are sometimes called the interface of a function as they make up the view of the function from the perspective of the caller.

```
double CircleCircumference(double radius)
{
      const double PI = 3.14;
      return (PI * (radius*2));
}
```

This has one formal parameter - `double radius`

The Actual Parameter List
The calling function must invoke the actual called function with some actual parameters.
The number of actual parameters must match the number of formal parameters.
The actual parameter type must match the formal parameter type, subject to default conversions.
Actual parameters and formal parameters can have different names or the same names. If Arguments and Parameters do have the same name, they are completely separate variables, and the parameters are used inside the function and are local to the function.

```
circumference = CircleCircumference(nextRadius);

double CircleCircumference(double radius)
{
      const double PI = 3.14;
      return (PI * (radius*2));
}
```

This has one actual parameter – `nextRadius`

Actual parameters are also referred to as arguments:
Arguments are variables that are defined in the calling module and are used within the function call.

Parameters are variables defined in the function header that receive the Argument values when the function is called.

Arguments are variables that are local to the calling function. Parameters are variables that are local to the function being called.

Matching Actual and Formal Parameters
If the formal parameters expect a string type to be passed the actual parameters must pass a string type - `void UserPrompt(string prompt)` – This function has

the formal parameter `string` so when we code a call to the function we must pass it a `string` type.

The matching between formal parameters and actual parameters is accomplished by order and not by type or name.

```
// In the calling function
double nextRadius, circumference;
string userMessage = "The circumference of your circle is ";
cin >> nextRadius;
circumference = CircleCircumference(nextRadius, userMessage);

// The called function
double CircleCircumference(double radius, string message)
{
      const double PI = 3.14;
      cout << message;
      return (PI * (radius*2));
}
```

In this example the function call would be ok as we pass the double first and the string second.

If we passed the string first and the double second we would generate errors - usually either parameter mismatch or linker

The return type, formal parameters and actual parameters define the communication possibilities open to a function.

If the Argument types are not of the same as their corresponding Parameters, then strange things can happen, but compiler / run-time errors may or may not be reported. Consider this example:

```
My_Function ('A');

void My_Function (int my_count)
{
      cout << my_count;
}
```

In this example, we have passed a character (e.g. 'A') as an argument for an integer parameter field, which results in the ASCII value for 'A' (i.e. 65) being assigned to the parameter. As a result, a value of 65 will be written to the screen due the argument / parameter type mismatch.

C++ is nice here, and does the conversion automatically for us, but this isn't always the case.

Default Parameters
A Default Parameter is a function parameter that is assigned a default value in the function header.

Once you supply a default value for a parameter, all subsequent parameters in the same Function Header or Function Prototype must also have default values. i.e. once you start giving default parameters, you must give all remaining parameters default values.

The following Function Headers are valid:

```
float Calculate_Y (float A_Val = 3, float B_Val = 5, float C_Val = 1,
float X_Val = 4)

float Calculate_Y (float A_Val, float B_Val, float C_Val = 1, float
X_Val = 4)
```

But this Function Header is invalid:

```
float Calculate_Y (float A_Val = 3, float B_Val = 5, float C_Val, float
X_Val)
```

because default values are assigned to *A_Val* and *B_Val*, but not to the remaining parameters *C_Val* and *D_Val*.

Function Declaration
A function name is an identifier so it must be formally declared before it is used.

The function declaration is essentially just a copy of the function header from the function definition.

```
// The whole thing is the function definition
double CircleCircumference(double radius, string message)
{     // this is the function body
      const double PI = 3.14;
      cout << message;
      return (PI * (radius*2));
}

// this is the function header
double CircleCircumference(double radius, string message);
```

The function declaration must specify the formal parameter types but not the formal parameter names.
It is however considered good practice to include the parameter name, it is also easier as you can copy and paste the header without modifying it.

Function declarations are typically declared to make them global in scope, although they don't have to be.

Function declarations are often referred to as prototypes.

This illustrates the typical initial arrangement of a program.

```
#include <iostream>
#include <string>
using namespace std;

// function prototypes – global in scope
int GetAge();
string GetName();
void TypeCast(const int value, char letter);

// constant - global in scope
const int DOG_YEARS = 7;

int main()
{
      // body of main
}

// function definitions then occur below main
```

Notes on functions
A quick re-cap:

- The return value and the parameter list are the methods of communicating with the function.
- When calling the function, you must use the same number and type of arguments in the correct order.
- If no return type is required for the function, then declare the function as a void function.
- The constant and variable declarations and program statements (C++ code) within a function must be framed within the parentheses type - curly brackets { and }.
- All variables and constants declared in a function are local to that function only and have no meaning outside of the function.

Value and Reference Parameters:

As stated above function parameters are the values passed to the functions. There are two types of function parameters:

Value Parameters (Pass-by-value)
You pass parameters to the function by value if they are used only for input to the function. These are called *Value Parameters*, and they provide **one-way communication** to the function.

By default (with the exception of arrays), C++ passes parameters by value.

A temporary allocation of memory is made for each of the formal parameters.

A copy of the value of the corresponding parameter is copied into that memory location.

The called function has no access to the actual parameter value just a copy of it.

That is, a copy of the actual argument's value is made into the particular parameter. That is, C++ automatically allocates memory to hold data of the argument's type, and then copies the argument's value into this memory location.

Value Parameters are treated exactly the same as local variables inside the function.

```
// Pass-by-value
void AlterNumbers(int number1, int number2)
{     // nothing would happen, changes
      // are not reflected back in
      // calling program
      number1++;
      number2--;
}
```

Reference Parameters (Pass-by-reference)
Reference Parameters are input and output parameters of the function and provide for **two-way communication** with the function.

To pass arguments by reference, use an "&" (ampersand) after the formal parameter type in the function definition and declaration

This causes the corresponding actual and formal parameters to refer to the same memory location. The formal parameters are therefore a synonym or alias for the actual parameter.

The called function can therefore modify the value of the actual parameter

You pass parameters by reference if they are used for input to the function and new values for these parameters are required back from the function. i.e. the values of the *Reference Parameters* can be used and changed by the function, and the changes made to these parameters are reflected back in the calling program.

Passing parameters by reference involves using a memory address as the actual parameter. For Reference Parameters, no memory is allocated, and the values of argument do not need to be copied.

```
// Pass-by-reference
void AlterNumbers(int& number1, int& number2)
{     // value changes are reflected
      // back in calling program
      number1++;
      number2--;
}
```

Constant Reference (Pass-by-constant-reference)
This refers to the passing of constants (those declared with the reserved word const)

The formal parameter type is preceded by the word const and follow by an ampersand (&)

Just like reference parameters the corresponding actual and formal parameters to refer to the same memory location.

However the called function is not allowed to modify the value of the parameter, the compiler will display an error message.

```
// pass-by-constant-reference
int AddTwoNumbers(const int& number1, const int& number2)
{     // the parameters number1 and number2
      // can not be modified
      return (number1 + number2);
}
```

**Note:* Reference and Value Parameters - If a function's parameter has an "&" (ampersand) prefix, then it is a Reference Parameter. If a function's parameter does not have an "&" (ampersand) prefix, and it is a basic data type (such as int, float, char) then it is a Value Parameter. However, for other complex data types (such as arrays) this is not so. Arrays, for example, are always passed to a function by reference - NOT by value - even if they do not have an "&" prefix.

**Note2:* Which is best value or reference – There are many pros and cons the main ones are covered here. The pros of reference are you get dynamic binding and flexibility. The pros of value are speed.

Parameter Restrictions

With pass-by-value the actual value can be an expression or literal constant as well as a variable or constant.

With pass-by-reference the actual parameter must be an *l-value*. That is something to which a value can be assigned.

```
// the function using
// expression for parameter one
F = CalcForce(mass * g, h);

// the function
double CalcForce(double Weight, double Height)
{
      // body of function
}


// the function using literal
// constant for parameter one
PrintString("Literal Constant");

// the function
void PrintString(string toPrint)
{
      // body of function
}
```

Choosing Between Pass-by- Methods
Pass-by-Reference only if the design of the program requires that it be able to modify the value of the parameter

Pass-by-constant-reference If the function has no need to modify the value of the parameter, but the parameter is large – a long string or array

Pass-by-Value considered the safest mechanism as no altering of values is going on. Use in all cases where the above aren't required.

** Note: Remember in C++ there is no well to tell by looking at a function call which values are passed by reference or value. It is considered better to have a return type than use pass-by-reference*

A Note on Static Variables

Each function is a discrete block of code.

The code and data defined within a function cannot interact with the code or data defined in another function, except via direct function calls.

Variables defined within a function are 'local' to that function, and have no meaning outside of the function. i.e. they have the scope of the function.

All *local variables* comes into existence when the functions are entered (executed) and are destroyed upon exit (completion) of the function.

The only exception to this rule is variables defined as *static*. These have the scope of the function but retain their value between invocations of the function.
As mentioned, *static variables* retain their value between function invocations. Consider this complete program:

```
#include <iostream>     // For cout.

void Static_Example();  // Function Prototype

int main()
{
      Static_Example();
      Static_Example();
      Static_Example();

      return      0;
}

void Static_Example()
{
      static int Count_Val = 0;     // Static, initial value = 0.
      std::cout << "My Count = " << Count_Val << std::endl;
      ++Count_Val;                  // Increment Count_Val.
}
```

Running this program would result in the following output:

My Count = 0
My Count = 1
My Count = 2

The value of the static variable (*Count_Val*) is retained and remembered between successive executions of the function dues to its *static* definition.

** Note: Static variables can be extremely useful.*
Always take careful note of the declarations of variables within a program, and in particular watch out for any static variables.
By simply adding a "static" keyword to the declaration of a variable, an enormous impact on the program can result.
As we move forward and begin to look at classes we will see how static can apply to member variables shared between classes.