C++
Functions

CO650 Advanced Programming

Topics

- Functions
- Parameters
- Default Parameter Values
- Declaration & Definition

Functions

- A named block of code that can optionally return a value.
- A building block for modular designed software

```
type   FunctionName(){
     statement/s;
}
```

- The name must conform to identifier naming rules.
- If no value is returned the type is void.
- Invoke the function using the name of the function followed by the call operator **()**

```
void PrintName(){
            cout << "Fred Bloggs" << endl;
}

int main(){
            PrintName();
}
```

Return Value

- If the function is returning a value then its type must match the type to be returned.
- The return keyword finishes the function call and returns the value to the right.

```
type   FunctionName(){
     statement/s;
     return value;
}
```

- Any statements placed after the return keyword will not be executed

Functions

```
int GetAge(){
        int age;
        cout << "Please enter your age ";
        cin >> age;
        return age;
}


int main(){
        int myAge = GetAge();
        cout << "Your age is " << myAge << endl;
        return 0;
}
```

Parameters

- Data in the form of parameters can be passed to a function.
- The parameters (formal parameters) are a comma delimitated set of types and identifier names placed within the function's ()
- Parameters are treated as local variables within the function.

```
type  FunctionName(type identifier1, type identifier2){
    statement/s;
}
```

```
int Add(int a, int b){
        return (a + b);
}

int main(){
        cout << "The sum is : " << Add(5,8) << endl;
        return 0;
}
```

Parameters Passing

- There are four parameter passing mechanisms
  1. By value
  2. By reference
  3. By constant reference
  4. By pointer

- Decision as to which to use should consider
  - Should the function be able to change the value of the variable passed to the parameter. If so use by reference or pointer.
  - If not and the type of the formal parameter is primitive use by value or in the case of a user defined type use constant reference.

Passing By Value

- A copy of the argument is assigned to the formal parameter.
- Should only be used when the overhead of making the copy is acceptable.
- Typically used for parameters of primitive types

```
type  FunctionName(type indentifier){   }
```

```
void AdjustMark(int mark, bool late){
        if (late && (mark > 40))
                mark = 40;
}

int main(){
    int theMark = 50;
    AdjustMark(theMark,true);
    cout << "Mark By Value : "<< theMark << endl;   // Displays 50!!
}
```

- Be careful- while mark's value changes to 40 theMark retains its value of 50.

Passing By Reference

- & indicates the formal parameter accepts a reference.
- theMark and mark share the same value. Any changes to mark will also change theMark
- Commonly used with user defined types as it avoids the copy overhead

```
type  FunctionName(type& indentifier){   }
```

```
void AdjustMark(int& mark, bool late){
        if (late && (mark > 40))
                mark = 40;
}

int main(){
    int theMark = 50;
    AdjustMark(theMark,true);
    cout << "Mark By Reference : "<< theMark << endl;   // Displays 40
}
```

Passing By Constant Reference

- **&** indicates the formal parameter accepts a reference.
- **const** indicates that the value pointed to by the reference can't be changed within the function.
- Typically used with user defined types

```
type   FunctionName(const type& indentifier){   }
```

This example will **NOT** compile as it attempts to change the value of a constant

```
void AdjustMark(const int& mark, bool late){
        if (late && (mark > 40))
                mark = 40;
}

int main(){
    int theMark = 50;
    AdjustMark(theMark,true);
    cout << "Mark By Constant Reference : "<< theMark << endl;
}
```

Passing By Pointer

- Mechanism used within C code and inherited by C++
- The address of a variable is passed as an argument to the formal parameter of type pointer.

```
type   FunctionName(type * indentifier){    }
```

- The pointer mark is assigned the address of theMark and therefore access the same value.

```
void AdjustMark(int * mark, bool late){
            if (late && ( *mark > 40))
                        *mark = 40;
}
int main(){
    int theMark = 50;
    AdjustMark( &theMark ,true);
    cout << "Mark By Pointer : "<< theMark << endl;   // Displays 40
}
```

Arrays As Parameters

- The array name is a pointer to the first element in the array.
- This can be passed as an argument to a function

```
type FunctionName(ArrayType* identifier)
```

- Note the size of the array is passed as well in this example. If not the loop would not know how many times it should iterate.

```
void Display( int* data , int size){
        for (int n=0; n < size; n++)
                cout << data[n] << endl;
}

int main(){
        int marks[3] = { 20,45,75 };
        Display( marks , 3 );
}
```

Multi Dimensional Arrays As Parameters

- Two or more dimensional arrays can't be passed as a simple pointer.
- The parameter accepting the array should be declared in the same way the argument is.

```
returnType FunctionName(ArrayType identifier[size1][size2])
```

- The first size may optionally be omitted.

```
void Display( int data[][4] , int size){
        for (int r=0; r < size; r++)
                for (int c=0; c < size; c++)
                        cout << data[r][c] << endl;
}
int marks[4][4];
int main(){
        Display( marks , 4 );
}
```

- marks and data point to the same array.

Constant Parameters

- By preceding a parameter with the **const** keyword we can ensure that its value is not changed within the function.

```
type FunctionName(const type ParameterName)
```

```
bool IsRetired(const int age){
    if (age >= 65)
        return true;
    else
        return false;
}
```

- In the case of a reference, the value the reference referrers to, can't be changed. The reference is a const anyway.

```
bool IsRetired(const int& age){
    if (age >= 65)
        return true;
    else
        return false;
}
```

Constant Parameters

- In the case of a pointer, the value the pointer points to can't be changed.

```
bool IsRetired(const int * age){
	if (*age >= 65)
		return true;
	else
		return false;
}
```

```
type FunctionName(Type const * ParameterName)
```

- To make a pointer const rather than the data it points to, place the const before the *

```
bool IsRetired(int const * age){
	if (*age >= 65)
		return true;
	else
		return false;
}
```

Default Parameters

- The last parameter/s can be assigned default values.
- If when invoking the function, the arguments are not passed, the default values will be assigned.

```
type FunctionName(type identifier = value)
```

- If an argument is passed the default will be ignored.

```
float CalculateTax(float income,float taxRate = 0.20){
        return (income - (income * taxRate));
}

int main(){
        cout << CalculateTax(32000) << endl;
        return 0;
}
```

- It is illegal to assign the first parameter a default and not the second.

Function’s Signature

- Comprises the function’s
  1. Name
  2. Number of parameters
  3. The parameter type.
- The return type is not part of the signature.

Function's Declaration

- Sometimes referred to as the prototype.
- Shows the function's interface. Not the statements within the body.
- Consists of Function name, parameter types and return type.
- Parameter names are optional.

```
type FunctionName(type,type etc);
```

- Often related function declarations are placed in a header file.

```
float CalculateTax(float,float);
```

Function’s Definition

- Includes the function declaration along with parameter names and body of code.

```
type FunctionName(type identifier , type identifier) {
    statment/s
}
```

```
float CalculateTax(float income, float taxRate){
            return (income - (income * taxRate));
}
```

Why Declare

- In C++ a function must have been declared or defined before it can be invoked up to that point within the code.
- It is possible to overcome this by declaring the function above the invocation within the code and then later defining it.

```
void AdjustMark(int * , bool);

int main(){
        int theMark = 50;
        AdjustMark(&theMark,true);
        cout << "Mark By Pointer : "<< theMark << endl;   // Displays 40
        return 0;
}

void AdjustMark(int * mark, bool late){
        if (late && (*mark > 40))
                *mark = 40;
}
```

Forward Declaration

- There are two approaches to centralising the location of global variables.
  1. In header files which can then be included
  2. In a cpp file.
- To use a global variable that has been declared in a cpp file you must declare the variable using the extern keyword.
- This tells the compiler that the variable has been declared in a different file.

```
// cpp file
int aGlobalVariable = 20;
```

```
// File in which you wish to access global
extern int aGlobalVariable;

int main(){
      cout <<  aGlobalVariable << endl;
      return 0;
}
```

Summary

- Unlike true Object Oriented languages functions can exist outside of a class in C++.
- Functions are the building blocks of procedural programming.
- Parameters improve the flexibility of functions.
- The const keyword can be used to avoid unwanted side effects.