Module 2: Modular Programming

Lesson 1: User defined functions and Parameter passing

Learning Outcomes

- LO 2.1.1 Implement value returning functions to simplify code clutter.
- LO 2.1.1 Pass non-default and default parameter to a function.

Creating user defined functions

When a complex problem would be dealt the code base would grow in size. That is why, C++ provides a mechanism similar to any other programming language that would create modules of the program called functions. These function contains a block of code that performs a specific task. All C++ programs contain at least one function – the `main ()` function. It is typically responsible for calling other functions although any function can call any other function. Function are generally used for the following reasons
Avoid task duplication
Make large tasks manageable
On Chapter 1 math functions are used to simplify the complex task of mathematical equations. A syntax and an example for creating a user defined function are a follows.

Table 1: Function Example

Syntax	Example

`functionPrototype`
`…`
`returnDataType`
`functionName(`*`parameterList`*`)`
`{`
`  statements`
`  …`
`  return (value)`
`}`
	`#include <iostream>`
	`using namespace std;`
	`void displaySum(int x, int y);`
	`double cmToInches(double x);`
	
	`int main()`
	`{`
	
	`  return 0;`
	`}`
	
	`void displaySum(int x, int y)`
	`{`
	`  cout<<x+y<<endl;`
	`}`
	
	`double cmToInches(double x)`
	`{`
	`  return x/2.54;`
	`}`

Where:

- `functionPrototype` – is a complete function declaration without a function body. This tells the compiler what the function has, the type of return value and the number of function arguments.
- `returnDataType` – is the datatype that would be given back once the function would finish its operation. However a `void` function would not have any return value.
- `functionName` - is the identifier used so that the function can be invoked (or called). Standard rules of naming identifiers would apply. However it is advisable to begin a function name with an action word.

- `parameterList` – also known as *formal parameters*, this would contain the information that would be given to function once it is invoked.
- `statements` - are the executable task bounded within curly braces.
- `return (value)` - is the actual value that is returned as a result of the process given by the function.

Void function

Placing a `void` instruction before a function name would nullify any form of data type with its corresponding value that is returned from a function. However placing a return statement (without any value) would terminate the operation of the function. The void function may have actual parameters but when invoked it must be a standalone statement. Meaning, they cannot be called in an expression because they contain no value. Using the code base on Table 1, the function call of a void function is done on function `main`.

Table 2: Calling a void function

Syntax	Example
functionName (arguments);	displaySum (4,7); //output: 11

An output is automatically displayed after the function is invoked. The dispSum function cannot be assigned to any variable since there is no return value.

Value Returning function

Placing any datatype before a function name would indicate that the particular function would return a value of that datatype. The value that a function would begin with (i.e. passed for processing) is known as the ***argument***. When a function has finished execution the ***value returned*** is the result of the processing. A function may have many arguments but only one value returned. By using the function on Table 1, calling a value returning function on function `main` is as follows:

Table 3: Calling a value returning function

Syntax	Example

optionalVariable = functionName (arguments);	double x = cmToInches(10);
	cout<<x;
	//output: 3.93701

The function `cmToInches` accept 10 as an argument, then the returned value is stored on variable `x`, before it is displayed as an output.

Parameter Passing

The parameter allows the sending of values to the function's internal operation. When a function is called the argument(s) that are given to a function is known as the ***actual parameter***. During execution, the actual parameter is accepted on the parameter block of the function, and is now called as the ***formal parameter*.**

Table 4: Parameter passing example

Example

```
double z = 10;
          cmToInches          (z);//actual
parameter
…                    //↓
double     cmToInches(double     x)//formal
parameter
{
    return x/2.54;
}
```

Pass by Value

The example on Table 4 follows the operations of passing a parameter by value (value parameters). That is, the content of the variable assigned on the actual parameter is copied to the variable on the formal parameter. Meaning that the changes made of the variables within the function is not reflected on the invoking function.

Example

```
    int y = 5;
    squareOfInt(y);
    cout<<y;
…
int squareOfInt(int y)
{
    y = y * y;
    return y;
}
//output: 5
```

Although integer `y` is passed to the `squareOfInt` function, and within the function the value of `y` is modified, it will not be reflected on the calling function since only the value of `y` is copied for the value of y on function `squareOfInt`.

Pass by Reference

On the other hand, passing by reference (reference parameters) would link the address of the variable from the calling function to the function that is invoked. So, any changes of that particular variable on the function would reflect to the calling function. This is done by placing an ampersand (&) after the datatype in the formal parameter.

Example

```
    int a = 3;
    cubeOfInt(a);
    cout<<a;
…
int cubeOfInt (int &b)
{
    b = b*b*b;
    return b;
}
//output: 27
```

The value of variable a in the calling function would now be modified since the address is referenced to the called function.

Passing by constant

When passing by constant the programmer would have the benefits of both variable sharing and disallowing changes with its content. This is done by placing the word const before the datatype.

Example

```
    int x = 10,y=5;
    cout<<productXY(x,y);
…
int productXY(const int &x, const int &y)
{
    /*x = x * y; cannot be since x
    is a read only variable instead:*/
    int z = x * y;
    return  z;
}
```

For this example, if the value of variable `x` would be modified, the compiler will automatically prompt an error indicating that `x` is a read-only variable.

Default parameter on functions

As you have observed, the number of actual parameters is the same with the number of formal parameter. C++ eases this condition by allowing the use of default parameters. When you specify a value of a variable on the function prototype this would act as the default value. Generally these are the rules for functions with default parameters.

- All default parameters must be at the rightmost parameters
- Default values would be used when no value is specified during a function call.

- However, when the caller would specify a value, the default value will be overridden.
- Default values can be constants, function calls, or global variables.
- A reference parameter cannot be assigned with a constant value.

Example

```
double defParamExp (int a, double b, int c=1, double
d=3.5);
…
    int x = 3;
    double z=1.5;
    cout<<defParamExp(1,2.5)<<endl;   \\output: -2
    cout<<defParamExp(x,z,7)<<endl;    \\output: -20
    cout<<defParamExp(x,1.1,7,z)<<endl; \\output: -4
…
double defParamExp (int a, double b, int c, double d)
{
    return (a+c)*(b-d);

}
```

The `defParamExp (1,2.5)` function call would assign the 1 and 2.5 value on `int a` and `double b` respectively and the default values for `int c` and `double d` would be used. The next function call, which is `defParamExp (x,z,7)` will pass the values of `x` and `z` for parameter `a` and `b`, and will override the default value of `c` by 7. The last function call would assign values for all parameters regardless if it is a default parameter or not.

Precautions in Parameter passing

To prevent programming errors you must apply the following rules.

- There must be the same number of actual parameters to formal parameters except for default parameter.
- To prevent data loss and subsequent error for overloaded function, the data type for each actual parameter must be the same with the datatype of formal parameter.
- When passing by reference the actual parameter must be a variable, whereas the actual parameter when passing by value may be a variable, constant or expression.

Example

```
void displaySum(int x, int y);
int cubeOfInt(int &b);
...
//erroneous function calls
displaySum(1);        //rule 1
displaySum (2.5, 3.3);//rule 2
cubeOfInt(1);         //rule 3
```

Lesson 2: Return values as Boolean Parameters, Scope and Lifetime of Variables and Overloading Functions

Learning Outcomes

- LO 2.2.1 Use returned values of function to control selection and repetition structure.
- LO 2.2.2 Trace the scope and lifetime of a variable
- LO 2.2.3 Overload functions using default parameters.

Using return values as Boolean Parameters of Control Structures

There are many cases in which complex statements are required to satisfy the logical expression on a control structure. To address this, a function with a Boolean return type can be used to replace the complex statement thus reducing the amount of code and the possibility of an error.

Example

```
bool isEven (int x);
…
    if (isEven(5))
        cout<<"Even";
    else
        cout<<"Odd";
//output: Odd
…
bool isEven(int x)
{
    if (x%2==0)
        return 1;
    else
        return 0;
}
```

Scope and Lifetime of variables

Scope – refers where the variable is used for a particular program. This can be global or local.

Lifetime – refers to how long a variable would remain in the internal memory of the computer.

Local variables has a scope to that particular function in which it is declared. So, when the function would finish execution, the variable would be released from the memory. On the other hand ***global variables*** are those variables that are declared outside any function. Since they can be used by any function, it would only be released from the memory once the program would end.

Example

```
int globalVar = 1;              //globalVar is a global variable
void displaySum(int x, int y);//x and y are local variables
```

Global variables must only be declared if this variable is used by most functions. Since, common errors would occur especially on the aspect of unintentionally changing the content of the variable.

Overloading Functions

Some function in C++ may have the same name but is unique on their formal parameter. This is also known as function overloading. To implement this you must have to remember these specific rules.

- They must have different number of formal parameters.
- If the number of formal parameters are the same, they must at least differ on the order of listing.

Example

```
int add(int x, int y);
double add(double x, double y);
char add(char x);
…
  cout<<add(1,2);<<endl      //output 3
  cout<<add(1.5,2.51);<<endl //output 4.01
  cout<<add('A');<<endl     //output a
…
int add(int x, int y)
{
    return x + y;
}

double add(double x, double y)
{
    return x + y;
}

char add(char x)
{
    return x + 0x20;
}
```

The first function call would add integers, so it would activate the `int add(int x, int y)` function. The second, would add floating values so, it would activate `double add(double x, double y)`. The last function call has only one parameter so, it would activate the `char add(char x)` function which would add a hexadecimal value of 20 to the value of the formal parameter.

Lesson 3: Recursion

Learning Outcomes

- LO 2.3.1 Modify iteration statement using recursion.

C++ has an interesting feature in which it a function can call itself. This ability is known as recursion. This feature is particularly useful when large problems would be reduced to smaller versions of itself. Thus, certain problems that are complicated to solve using traditional iteration can easily be addressed using recursion. Moreover, artificial intelligence usually employ the capability of recursion. However, for this chapter only an introduction to recursion would be covered.

Recursive function

In a recursive function, the function will call itself until a certain parameter would reach its base case (limit). Typically this is done by including the recursive call in an if statement. Then reducing the general case (parameter) of the function until it would reach its base case.

General Form	Example

`datatype functionName (parameter)`
`{`
`  statements`
`  if (!limit)`
`  return functionName (reduce Parameter)`
`}`
	`int factorial(int x)`
	`{`
	`  if (x>1)`
	`  {`
	`  return x * factorial(x-1);`
	`  }`
	`  else`
	`  {`
	`  return 1;`
	`  }`
	`}`

Figure traces the process of factorial implemented through recursive function.

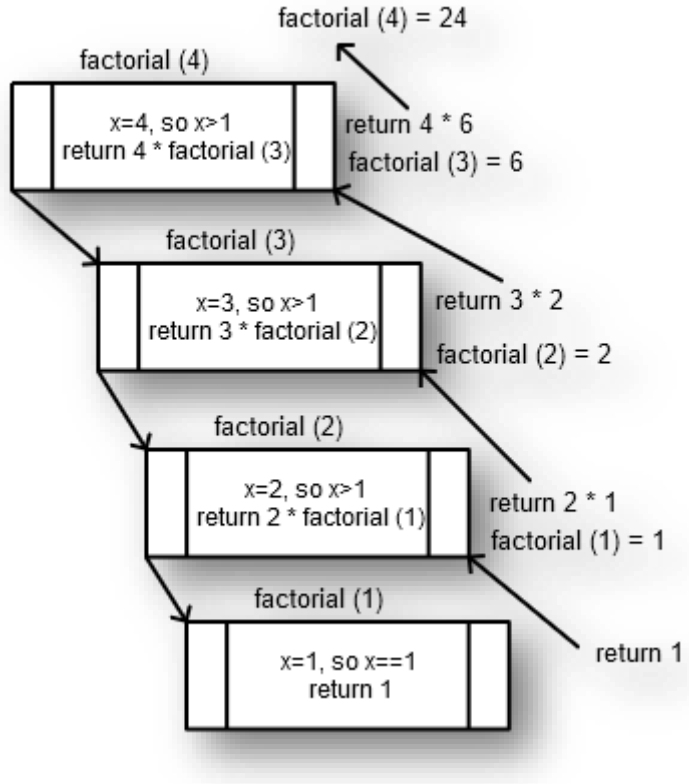

Figure 1: Process flow of the recursive factorial

Recursion or Iteration?

There are generally two ways to solve a problem with repeated operations. One is to use iterative statements such as while, for or do… while, and the other is to

implement recursive function controlled via if statement. However it is important to remember that a recursive function would automatically allocate local variables for each function call. Only when the function is terminated is the memory for the local variables deallocated. This would in turn cause a high overhead of memory and processing time compared to iteration. So does this mean that iteration is better? Not exactly since the computers today are having inexpensive memory and fast processors making the overhead unnoticeable. Furthermore there are a lot of problems especially those with divide and conquer strategy that are far easier to implement in recursion rather than iteration. However, problems that uses dynamic programming strategy may have the same level of difficulty if it is implemented in both recursion and iteration, yet an iterative solution is many times faster than a recursive solution. Therefore, if the performance is your concern, better use iteration, but if performance is not an issue and recursive solution is obvious, use recursion.