6. User Defined Functions I

Functions are like building blocks. They are often called **modules**. They can be put togetherhe to form a larger program.

Predefined Functions

To use a predefined function in your program, you need to know only how to use it. Every function has a name and, depending on the value specified by the user, it does some computation. The type i.e. `double` is specified by the function. The values are called the **parameters** (or **arguments**). Because the value of the function `pow` is of the type `double`, we say that the function `pow` is of the type `double`.The `x` and `y` are its arguments. In C++, predefined functions are organized into separate libraries i.e. `iostream` or `cmath`. To use predefined functions in a program, include the relevant header file.

User-Defined Functions

User-defined functions in C++ are classified into two categories.
• Functions that have a data type, called **value-returning functions**
• Functions that do not have a data type, called **void functions**

Value-Returning Functions

The name of the function, the number of the parameters and their type and the data type returned form the **heading** of the function is also called the **function header**. The code to accomplish the task is called the **body** of the function. Together they form the **defination** of the function. A variable declared in the function heading is called **formal parameter**. A variable or expression listed in a call to a function is called **actual parameter**. The syntax of a value returning function is:

```
functionType functionName(formal parameter list)
{
    statements
}
```

The `functionType` is also called the **data type** or **return type**. The statements enclosed between curly braces form the **body** of the function. The syntax of the formal parameter list is:

```
dataType identefier, dataType identifier, ...
```

The syntax to call is:

```
functionName(actual parameter list)
```

The syntax for actual parameter list is:

```
expression or variable, expression or variable, ...
```

Formal parameter list can be empty. Even then the parentheses in the heading and the call are required. Actual and formal parameters have a one-to-one correspondence. A function call in a program causes the body of the called function to execute.

`return` Statement

Once the value-returning function computes the value, it returns the value via the return statement.

```
return expr;
```

Where `expr` is a variable, constant value or expression. The data type of the value that `expr` computes must match the function type. When a `return` statement executes, the function immediately terminates and the control goes back to the caller. In `main` function, the `return` statement terminates the program. In a function call you specify only the actual parameter(s), not its data type. Once a function is written, you can use it anywhere in the program.

Function Prototype

Logically user-defined functions must be placed before `main`. Typically they are at the end. A function prototype is placed before `main`. **Function prototype** is the function heading without the body:

```
functionType functionName(parameter list);
```

Semicolons are necessary. The variable name is not necessary. Functions can appear in any order provided the function prototype is declared first. First statement in function `main` executes first. Other functions execute only when they are called.

7. User Defined Functions II

Void Functions

The functions that do not have a data type are called **void functions**. The `functionType` is void. The `return` statement can be used to exit early. The functions may or may not have formal parameters. The general syntax of the void functions without parameters is as follows:

```
void functionName()
{
    statements
}
```

Use meaningful name for `functionName`. Function call is:

```
functionName();
```

The communication link between the calling function and the called function is established by using parameters. The function definition of void functions with parameters has the following syntax:

```
void functionName(formal parameter list)
{
    statements
}
```

The formal parameter list has the following syntax:

```
dataType& variable, dataType& variable, ...
```

The `&` is optional and has special meaning discussed later. The function call has the following syntax:

```
functionName(actual parameter list)
```

Where actual parameter list is:

```
expression or variable, expression or variable, ...
```

There are two types of formal parameters: **value parameters** and **reference parameters**. Value parameter receives a copy of the content of the corresponding actual parameter. The reference parameter receives the location (memory address) of the actual corresponding parameter. It is done by attaching `&` after `dataType` in formal parameter list.

Value Parameters

When a function is called, the value of the actual parameter is copied into the corresponding formal parameter. if the formal parameter is a value parameter, then after copying the value of actual parameter, there is no connection between the formal parameter and actual parameter. The formal parameter manipulates the data stored in its own memory space. Value parameters can not pass information outside the function.

Reference Parameters

Because a reference parameter receives the address of the actual parameter, reference parameters can pass one or more values from a function and can change the value of the actual parameters. they are used to return more than one value from function. They can change the value of the actual parameter and can save memory space and time. If a formal parameter is a non-constant reference parameter, during function call its corresponding actual parameter must be a variable.

Value and Reference Parameters and Memory Allocation

When a function is called, memory for its formal parameters and variable declared in the body of the function (called local variables) is allocated in the function data area. In case of value parameter, the value of actual parameter is copied into the memory call of its corresponding formal parameter. In case of reference parameter, both the actual and formal parameters refer to the same memory location. Stream variable (i.e. `ifstream`) should be passed by reference to a function.

Reference Parameters and Value-Returning Functions

You can use reference parameters in a value-returning function, but is not recommended. If function needs to return more than one value, change it into a void function and use appropriate reference parameters.

Scoppe of an Indentifier

Identifiers are declared in a function heading, within a block, or outside a block.The **scope** of an identifier refers to where in the program an identifier is accessible (visible). The identifiers declared within a function (or block) are called **local identifier** They are not accessible outside of the function (or block). **Global identifier** are declared outside of every function definition. C++ does not allow the nesting of functions. There are different rules for global and local identifiers. The scope of a function name is same as the scope of a global variable. C++ allows to declare a variable in the initialization statement of `for` loop:

```
for (int count = 1; count < 10; count++)
    cout << count << endl;
```

The scope of the variable `count` is limited to only the body of `for` loop. In C++, `::` is called the **scope resolution operator**, used to access global variable in some situations i.e. `::z`. To use global variables declared after function definition, use **external variable**:

```
extern int w;
```

Memory for global variables remain allocated as long as the program executes. Memory for local variables is allocated at block entry and deallocated at block exit. Avoid using global variables as any function using global parameter is not independent.

Static and Automatic Variables

A variable for which memory is allocated at block entry and deallocated at block exit is called an **automatic variable**.By default variables declared within a block are automatic variables. A variable for which memory remains allocated as long as the program executes is called a **static variable** such as global variables. You can declare a static variable within a block by:

```
static dataType identifier;
```

Most compilers initialize static variable to their default values.

Function Overloading: An Introduction

Overloading refers to the creation of several functions with the same name. Every function must have different formal parameters or/and data types. The type/number of parameters determine which function to execute. Function overloading is used when you have the same action for different set of data.

Functions with Default Parameters

The default parameters are specified when function name appears for the first time such as in prototype. If you do not specify the value of default parameters, the default value is used. They must be right most. There are some other rules too. A typical example is:

```
void functionExp(int x, int y, char z='A', double w=78.34)
```

Here `z` and `w` are default. If you do not specify any values, their default values will be used.

8. User-Defined Simple Data Types, Namespaces and the `string` Type

Enumeration Type

To define an **enumeration type**, you need the name for the data type, a set of values and set of operation on the values. The operations may be avoided. The values must be identifiers. The syntax is:

```
enum typeName value1, value2, ...;
```

Where `value1, value2, ...` are identifiers called **enumerators**. The list implies the ordering i.e. `value1 < value2 < value3 < ...`. The default value start from 0. Enumerators are not variables. A typical example is:

```
enum colors {brown, blue, red, green, yellow};
```

Once a data type is defined, you can declare variable of that type as follows:

```
dataType identifier, identifier,...;
```

Once variable is declared you can store values in it:

```
popularColor = red;
myColor = popularColor;
```

No arithmetic operations are allowed on the enumeration type including increment and decrement operations. Use cast operator as follows:

```
popularColor = static_cast<colors>(popularColor + 1);
```

It advances `colors` by 1 in the list. Now it is `green`. The relational operators can be used with the enumeration type i.e.

```
red <= yellow is true
```

You can use enumeration type in loops. It increases readability. The enumeration type can be neither input nor output (directly). You can use `char` data type to read indirectly. If you try to output the value of an enumerator directly, the computer will output the value assigned. You can pass the enumeration type as a parameter to functions by either value or reference. You can declare variables of an enumeration type when you define an enumeration type. For example:

```
enum grades {A, B, C, D, F} courseGrade;
```

A data type wherein you directly specify values in the variable declaration with no type name is called an **anonymous type**. For example:

```
enum {basketball, football, baseball, hockey} mySport;
```

To avoid confusion first define an enumeration type and then declare variables.
The `typedef` statement is used to create synonyms or aliases to a previously defined data type. The general syntax is:

```
typedef existingTypeName newTypeName;
```

Namespaces

They are use to solve the problem of overlapping global identifiers. The general syntax is:

```
namspace namespace_name
{
    members
}
```

where a member is usually a named constant, variable declaration, function, or another `namespace`. The scope of a `namespace` member is local to the `namespace`. Use `::` to access the member:

```
namaspace_name::identifier
```

To simplify access of all `namespace` members

```
using namespace namespace_name;
```

To access a specific `namespace` member.

```
using namespace_name::identifier;
```

1 `string` Type

The data type `string` is a programmer-defined type and is not part of the C++ language. The library `<string>` supplies it. A string is a sequence of zero or more characters. The position of the first character in the string variable starts from zero. The binary operator + is used for concatenation One operand must be a string variable. For example:

```
    string str1, str2;
    str1 = "Sunny";
    sstr2 = str1 + " Day";
```

The **array index/subscript** operator `[]` denotes the position. For example the statement:
`str[6] = 'B';`
replace the 7th in `str` variable with `B`.

Additional `string` Operations

The data type `string` has a data type, `string::size_type`, and a named constant, `string::nops`, associated with it.The `length` has returns the number of characters currently in the string as an unsigned integer:

```
strVar.length()
```

The function `size` return the same value as `length`:

```
strrVar.size()
```

The `find` function searches a string to find a first occurrence of a particular and returns an unsigned integer of type `string::size_type`. The syntax is:

```
strVar.find(strExp)
```

or:

```
strVar.find(strExp, pos)
```

If search is unsuccessful, the function returns the special value `string::nops`.The `substr` function returns a particular substring of a string:

```
strVar.substr(expr1, expr2)
```

Where `expr1` specifies the position within the string and `expr2` the length of substring. The `swap` function is used to swap the contents the two string variables. The syntax is:

```
strVar1.swap(strVar2);
```

9. Arrays and Strings

A data type is called **simple** if variable of that data type can store only one value at a time. In **structured data type**each data item is collection of other data items. Simple data types are building blocks of structured data types.

Arrays

An **array** is collection of fixed number of components all of the same data type. A **one dimensional array** is an array in which the components are arranged in a list form.The general syntax to declare a one dimensional array is:

```
dataType arrayNsme[intExp];
```

Where `intExp` is any constant expression that evaluates to a positive integer. Also `intExp` specifies the number of the components. The general form used for accessing an array component is:

```
arrayName[indexExp]
```

where `indexExp`, called the **index**, is an expression whose value is a non-negative integer. In C++, `[]` is an operator, called the **array subscripting operator**. The array index starts at 0.When you declare an array, its size must be known. The array processing is done by stepping through the element of an array using a loop. For example:

```
int list[100];
int i;
for (i = 0; i < 100; i++)
    //process list[i]
```

If either `index < 0` or `index > arraySize - 1`, then we say that the `index` is **out of bounds**. This situation can result in altering or accessing unintended memory locations. Arrays can be initialized during declaration:

```
double sales[5] = {12.25, 32.50, 16.90, 23, 45.68};
```

The size can be omitted and determined by the number of elements. The `[]` are necessary. Arrays can be initialized partially:

```
int list[10] = {0};            //all elements are zero.
int list[10] = {8, 5, 12};     //first 3 are given, rest 3 are zero.
```

The copy, read, printing and comparing must be done component wise. In C++, arrays are passed by reference only as parameters to a function. For one dimensional array as formal parameter, size is usually omitted. You can prevent the function from changing the actual parameter by using keyword `const` in declaration:

```
void example(int x[], const int y[], int sizeX, int sizeY)
{
      .
      .
}
```

The **base address** of an array is the address (i.e., memory location) of the first array component. When you pass an array as a parameter, the base address of the actual array is passed to the formal parameter. C++ does not allow functions to return a value of the type `array`. Other than integers, C++ allows any integral type to be used an array index such as `enum`.

C-strings (Character Arrays)

An array whose components are of type `char` are called **character array**. In C++, the null character is represented as `'\0'`. The most commonly term used for character arrays is `C`-strings. `C`-strings are null terminated. `C`-strings are stored in one-dimensional character arrays:

```
char name[16] = "John"
```

C++ provides some functions like `strcpy(s1, s2)`, `strcmp(s1, s2)` and `strlen(s)`. To use these functions include the header file `cstring` via the `#include` statement. `C`-strings are compared character-by-character using the systems' collating sequence. C++ allows aggregate operations on array as input/output of `C`-strings.

```
char name[31];
cin >> name;
```

`C`-strings that contain blanks can not be read using >>. Use `get` instead.

```
cin.get(str, m+1;)
```

It stores next `m` characters or until `'\n'`. You can use `cout` to output string.

```
cout << name;
```

C++ allows the user to a specify the names of the input/output files at the execution time. You have to use `C`-string and not the `string` type. The header file `string` contains the function `c_str` for conversion. The syntax is:

```
strVar.c_str()
```

Two (or more) arrays are called **parallel** if their corresponding components hold related information or data.

Two-Dimensional Arrays

Two-dimensional array is a collection of a fixed number of components arranged in rows and columns, wherein all components of the same type. The general syntax is:

```
dataType arrayName[intExp1][intExp2]
```

The two expressions specify the number of rows and no of columns. To access the components you need a pair of indices:

```
arrayName[indexExp1][indexExp2]
```

Two-dimensional array can be initialized when they are declared:

```
int board[2][3] = {{2, 3, 1},{9, 4, 7}}
```

At least on of the values should be given to initialize all the components.You can also use enumeration type for array indices. For example:

```
inStock[Ford][White] = 15;
```

A two-dimensional array can be processed entirely, row-wise or column-wise. When processing a particular row or column, we use algorithms similar to one-dimensional array. For example to process column number 2 i.e.

```
for (row = 0; row < numberOfRows; row++)
    process matrix[row][2]
```

Use nested for loop to process an entire array.

```
for (row = 0; row < numberOfRows; row++)
    for (col = 0; col < numberOfCols; col++)
        arrayName[row][col] = 0;
```

Two-dimensional arrays can be passed as parameter to a function and they are passed by reference. The base address is passed to a formal parameter When declaring a two dimensional array as a formal parameter, you can omit the size of first parameter, but not the second one i.e.

```
void sumRows(int matrix[][numberOfCols], int numberOfRows)
{
    .
    .
}
```

Strings in C++ can be manipulated using either the data type `string` or character arrays (C-strings). For example:

```
string list[100];
char list[100][16];
```

You can use `typedef` to first define a two dimensional array data type and then declare variables of that type:

```
typedef int tableType[20][10];
tableType matrix;
```

Multidimensional Arrays

A collection of a fixed number of elements (called components) arranged in n dimensions ($n \geq 1$) is called an **n-dimensional array**. The syntax is:

```
dataType arrayName[intExp1][int Exp2] ... [intExpn]
```

Accessing is similar:

```
arrayName[indexExp1][indexExp2] ... [indexExpn]
```

When declaring a multi dimensional array as formal parameter in a function you can omit the size of the first dimension but not other dimensions.

10. Records (structs)

C++ provides a structured data type called struct to group items of different data type. An array is an homogeneous data structure; a struct is typically a heterogeneous data structure. Using struct, a single variable can pass all the components as parameters to a function. A struct is a collection of a fixed number of components in which the components are accesses by name. Th components may be of different types. The components of a struct are called the **members** of the struct. The general syntax is:

```
struct structName
{
    dataType1 identifier1;
    dataType2 identifier2;
      .
      .
    dataTypen identifiern;
};
```

Like any data type, a struct is a definition, not a declaration. No memory is allocated. Once a data type is defined you can declare variables of that type:

```
structName identifier1;
```

To access a structure member (component), you use the struct variable name together with the member name; these names are separated by a dot (period), called **member access operator**. The syntax is:

```
structVariableName.memberName
```

The structVariableName.memberName is just like any other variable. We can assign the value of one struct variable to another struct variable of the same type by using an assignment statement. To compare struct variable of the same type, you compare them member-wise. You cannot use relational operators on struct variable. No aggregate input/output operations are allowed on a struct variable. Data in a struct variable must be read one member at a time. Similarly the contents of a struct variable must be written one member at a time.
A struct variable can be passed as a parameter either by value or by reference, and a function can can return a value of type struct. A list has two things associated with it: the values/elements and the length. We can define struct containing both items.

```
const arraySize = 1000;

struct listType
{
    int listElement[arraySize];
    int listLength;
};
```

We can use arrays to process struct data. i.e.

```
employeeType employees[50];
```

This statement declares an array employees of 50 components of the type employeeType. Every component of employees is a struct.
The members of a struct can be of type of another struct. This can be easy to manage.The struct can be used to build another struct. Use member access operator multiple times i.e.

```
newEmployee.name.first = "Mary";
```

A struct can be a member of another struct.

16. Recursion

The process of solving a problem by reducing it to smaller version of itself is called recursion.

Recursive Definitions

A definition in which something is defined in terms of a smaller version of itself is called **recursive definition**. Every recursive definition must have one (or more) base cases. The general case must be eventually reduce to a base case. The base case stops the recursion. An algorithm using recursive definition is called **recursive algorithm**. A function that calls itself is called a **recursive function**. Recursive algorithms are implemented using recursive functions. For example:

```
int fact(int num)
{
    if (num == 0)                //base case.
        return 1;
    else
        return num * fact(num - 1);  //general case
}
```

A function is called **directly recursive** if it calls itself. A function that calls another function and eventually results in the original function call is said to be **indirectly recursive**. A recursive function in which the last statement executed is the function call is called a **tail recursive function**. If every recursive call results in another recursive call, then the recursive function (algorithm) is said to have infinite recursion. Make sure that every recursive call eventually reduces to a base case.

Problem Solving using Recursion

To design a recursive function you must do the following.

1. Understand the problem requirement.
2. Determine the limiting conditions i.e. no of elements in the list.
3. Identify the base cases and provide a direct solution to each base case.
4. Identify the general cases and provide a solution to each general case in terms of smaller version of itself

Recursion or Iteration

The **iterative control structure** use a looping structure, such as `while`, `for` or `do ... while`, to repeat a set of statements. In recursion a set of statements is repeated by having the function call itself. Moreover, a selection control structure is used to control the repeated calls in recursion. There is overhead associated with executing a recursive function both in terms of memory space and computer time. The choice is determined mainly by the nature of the problem. If the definition of a problem is inherently recursive, then you should consider a recursive solution.