Lecture 3.1

Topics

1. Functions With No Arguments
2. Modularization – Brief
3. Functions – Continued
4. Basic Computer Programming Structures

---

1. Functions with No Argument

One can use a function to group several statements together and then call the function as needed.

The simplest form of function has the following prototype,

```
void functionName(void);
```

This form of function will return no value and will require no argument. An example is drawn from assignment as follows,

Example 1

```
/**
 *Program Name:  cis26L0311.cpp
 *Discussion:    Simple Function
 */
#include <iostream>
using namespace std;

//Function prototype
void printClassInfo(void);

int main() {
  printClassInfo();

  return 0;
}

/**
 *Function Name: printClassInfo()
 *Description:   To print class information
 *Pre        :   Nothing (no argument required)
 *Post:      :   Displaying class information on screen
 */
void printClassInfo() {
  cout << "Class Information --"
       << "\n  CIS 25 -- C++ Programming"
       << "\n  Laney College" << endl;
  return;
}
```

OUTPUT

```
Class Information --
  CIS 25 -- C++ Programming
  Laney College
```

Note!

1. Function should be declared through its prototype, then
2. Function must then be defined – Function Definition, then
3. Function can be used or called

2. Modularization – Brief

Modularization is essential and frequently employed in software and applications. One should understand modularization as follows,

> *In a general top-down design, a program is divided into a main module and its related modules. Each module is in turn divided into submodules until the resulting modules are in the simplest form (or, intrinsic where they can be understood without further division).*

The concept is depicted in the so-called **structure chart** in **Figure 1** below. Note that in a structure chart:

- A module can be called by one and only one higher module as indicated by the arrows.
- A module or function can be written for reuse many times.
- A module may be called by another module and even itself.

In many cases, a structure chart represents an organized execution within the modules. It only shows function (module) flow and contains no code. The execution flow of these functions is determined by the following two rules:

(1) The execution should be read from top and going down, and
(2) On each level, it should start from left to right.

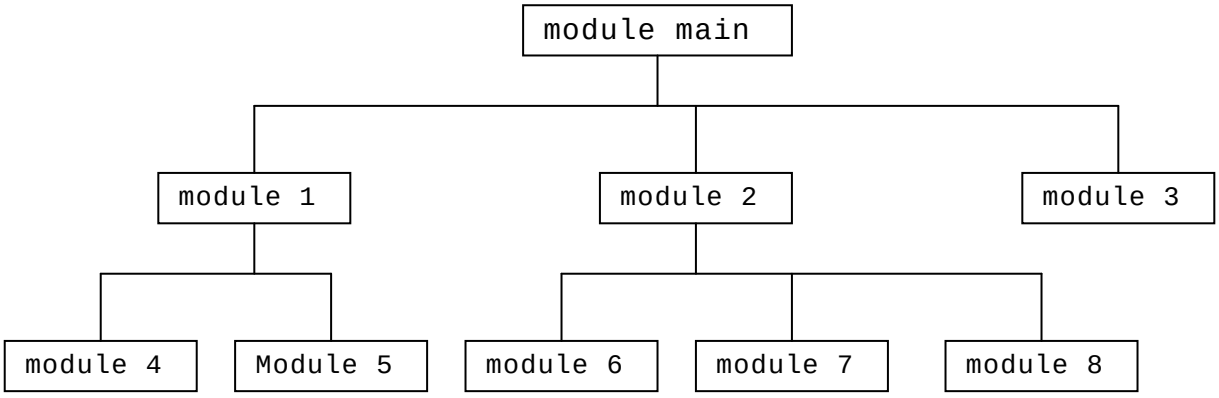

Figure 1 Structure chart

In the above structure chart, the execution flow will have the order as follows,

start(module main) → module 1 → module 4 → module 5 → module 2 → module 6 → module 7 → module 8 → module 3

More discussion on structure chart will be given during the semester. For now, the question is about module. What is module? What can it do? How does it get created? How can one use it?

3. Functions — Continued

In C++, module may be implemented through function.

- In general, a function is designed to be reusable.
- Preferably, a function must be designed to handle one task at a time.
- One or more of these functions may be called in another function to perform a more complex task.

When functions are used, the following processes may occur:

- A called function may receive values (i.e., parameters or argument values) from the calling function.
- The called function may return one and only one value to the calling function.
- During the execution of a function, it may produce side effects, which change the state of the program. Side effects may involve data from outside of the program, sending data to external files (monitor included), or variable values.

Let's look at **function declaration (**or, **prototype)**, **definition** (or, **implementation**), and **call (**or, **invocation/use)**.

3.1 General Definition

A function (method) in C++ is a self-contained unit of program in which its code was designed to accomplish a specific task. Writing code with function calls is essential and important in almost all programs and applications.

3.2 Function Prototype (Declaration)

```
ReturnType functionName( Type1 arg1, Type2 arg2 );
```

where

`ReturnType` is the type of the value to be returned by the function.
`functionName` is the name of the function.
`Type1` is the type of the argument by the name of `arg1`.
`Type2` is the type of the argument by the name of `arg2`.

3.3 Function Definition

The actual operations of the function must be defined in the body of the function as depicted below,

```
ReturnType functionName( Type1 arg1, Type2 arg2 )
{
  /*Function statements*/
}
```

Definition of Function Components

- The top line is called the **function header**. It must contain a return type and a function name followed by a pair of parentheses.
- Enclosed inside the parentheses, there may be an argument (parameter) list that has zero or more **`formal arguments`** (parameters). If there are formal arguments, each argument must be specified by a formal name and its type.
- The function body should have all operations and expressions defined.
- One should try to keep the function body as specific (i.e., one task per function) as possible. The function body should have minimal number of statements (e.g., approximately 22 statements/lines per function!).
- Components declared inside the function should only use internal data as much as it can. The function should minimize its interference to elsewhere outside its function body.

3.4 Prototypes – Digression

From the general prototypes, there are four groups of functions. The sample prototypes below show the differences between these four groups.

```
void functionGroup1(void);      // no return value
                                  // no argument

void functionGroup2(int, int);  // no return value
                                  // having argument(s)

int functionGroup3(void);       // having return value
                                  // no argument

int functionGroup4(double);     // having return value
                                  // having argument(s)
```

Example 2

```
/**
 *Program Name:  cis25L0312.cpp
 *Discussion:    Simple Functions
 */

#include <iostream>
using namespace std;

void printClassInfo(void);

void evaluateExpr(void);

bool isEven(void);

int main() {
  printClassInfo();

  cout << "\nCalling evaluateExpr():" << endl;
  evaluateExpr();

  cout << "\nCalling isEven():" << endl;
  if ( isEven() ) {
    cout << "\nThe number is even!" << endl;
  }
  else {
    cout << "\nThe number is odd!" << endl;
  }

  return 0;
}

/**
 *Function Name: printClassInfo()
 *Description:   To print class information
 *Pre        :   Nothing (no argument required)
 *Post:      :   Displaying class information on screen
 */
void printClassInfo() {
  cout << "Class Information --"
       << "\n  CIS 25 -- C++ Programming"
       << "\n  Laney College" << endl;
  return;
}

/**
 *Function Name: evaluateExpr()
```

```
 *Description:   To evaluate expression
 *Pre        :   Nothing (no argument required)
 *Post:      :   Displaying results of expressions
 */
void evaluateExpr() {
  int iVar;

  cout << "\nEnter an integer + ENTER : ";
  cin >> iVar;

  cout << "\n\t(1) Value of iVar : " << iVar << endl;
  cout << "\n\t(2) Value of iVar + 4 : " << iVar + 4 << endl;
  cout << "\n\t(3) Value of iVar = iVar + 4 : "
       << ( iVar = iVar + 4 ) << endl;
  cout << "\n\t(4) Value of iVar : " << iVar << endl;

  return;
}

/**
 *Function Name: isEven()
 *Description:   Asking for an int and evaluating its even/odd
 *Pre        :   Nothing (no argument required)
 *Post:      :   Returning true if even and false if odd
 */
bool isEven() {
  bool bValue = false;

  int i1;

  cout << "\nEnter an integer + ENTER : ";
  cin >> i1;

  if (i1 % 2 == 0) {
    bValue = true;
  }

  return bValue;
}
```

OUTPUT

```
Class Information --
  CIS 25 -- C++ Programming
  Laney College

Calling evaluateExpr():

Enter an integer + ENTER : 5

        (1) Value of iVar : 5

        (2) Value of iVar + 4 : 9

        (3) Value of iVar = iVar + 4 : 9

        (4) Value of iVar : 9

Calling isEven():

Enter an integer + ENTER : 4
```

```
The given number is even!
```

4. Basic Computer Programming Structures

In general, a programming structure is a basic unit of programming logic. There are three basic programming structures – (1) sequence (or step), (2) selection/decision, and (3) repetition/loop. A program may combine these three structures to produce the solution logic for a particular problem.

4.1 Sequence (Step) Structure

A sequence structure will provide programming operations or events in sequence one after another or step by step. This is depicted in **Figure 1a**, where each step or module can be just a single operation or several operations combined.

4.2 Selection or Decision Structure

A selection or decision structure is depicted in **Figure 1b**. There is a set of expressions to be verified before the next operation can be followed. In the basic structure, there are only two options to be considered as the outcomes of a decision. If the outcome represents a true then the flow will continue with one path. If the outcome represents a false then the flow will follow the other alternative path.

4.3 Loop/Repetition Structure

The loop structure is depicted in **Figure 1c** where the decision is checked before a selected event can be performed or followed. There are different variations of loop structure. We will revisit them in later lectures.

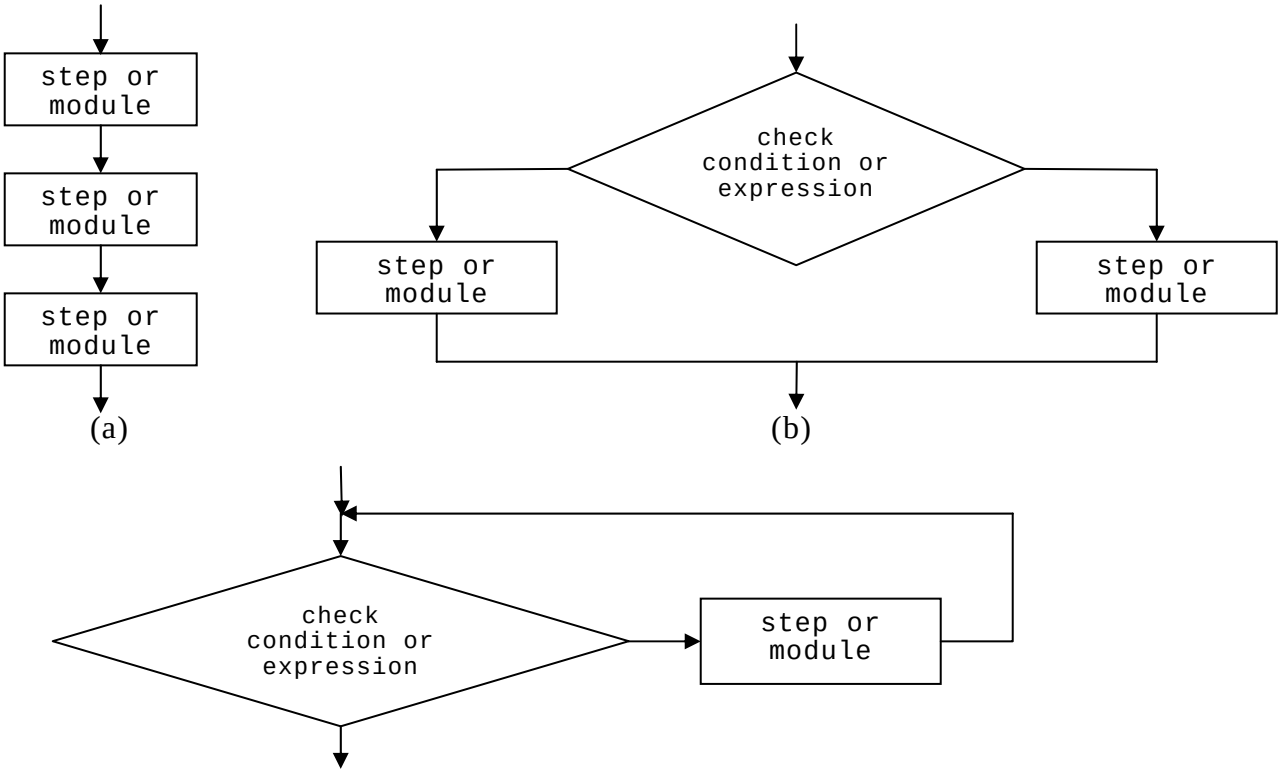

Figure 1 Three basic programming structures