Lecture 5.1

Topics

1. Modularization – Brief
2. Functions – Brief Introduction

---

1. Modularization – Brief

Modularization is almost always needed and 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 **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.

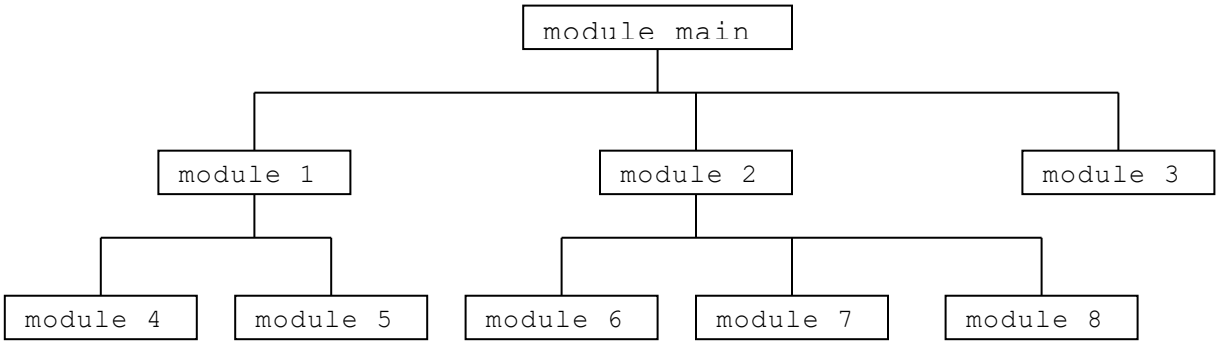

Figure 1 Structure chart

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.

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

module main → module 1 → module 4 → module 5 → module 2 → module 6 → module 7 → module 8 → module 3

More on structure chart will be given during the semester. For now, the questions are about module:

- What is function/method/module?
- What can it do?
- How does it get created?
- How can one use it?

Let's start to look at function/method/module next.

2. Functions – Introduction

In C a module is implemented as a function. A function should be designed to handle one task at a time. These functions may be grouped and used in another function to perform a more complex task.

A function in C is a complete and self-contained unit of code, which is designed and implemented to accomplish a specific task.

A function is designed to be reused during a program execution. When functions are used, the following processes may occur:

- A called function will receive control (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.

In general, working with a function would involve things such as

- Statement definition,
- Function declaration (prototype), and
- Function implementation (definition).

2.1 General Problem/Statement Definitions

In any program, a specific task may be required. This task is described and its solution is designed and solved through a set of operations. Then a function is one of the options to provide the implementation for this solution.

2.2 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 (parameter) by the name of `arg1`.

`Type2` is the type of the argument (parameter) by the name of `arg2`.

2.3 Function Definition (Implementation)

The function definition has the implementation for all steps and operations. The general syntax may be given as follows,

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

  /*Function statements*/

}
```

Definition of Function Components

- The top line is called the **function header**. It must contain a function name followed by a pair of parentheses, and a possible return type.

- Enclosed inside the parentheses, there may be an argument (parameter) list that has zero or more **formal arguments** (or formal 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.

2.4 Examples

The examples given below have several functions. They are declared and defined in the program, and then used (or called) in **main()**.

Example 1

```
/**
 * Program Name: cis6L0511.c
 * Discussion:   Functions with no arguments
 */
#include <stdio.h>

/*Function prototypes*/
void printClassInfo(void);

void printSquare(void);

void printSumTwoInt(void);

/*Application driver*/
int main() {

  printClassInfo();

  printSquare();

  printSumTwoInt();

  return 0;
}

/* Function definitions*/

/**
 * Function Name: printSquare()
 * Description:   Computing and displaying the square of
                  an integer
 * Pre:           Nothing
 * Post:          Displaying square value
 */
void printSquare() {
  int iValue;
  printf("\n  Computing square of int -- printSquare():\n"\
```

```
    "\tEnter an integer + ENTER: ");
  scanf("%d", &iValue);

  printf("\n\tThe square of %d is %d\n", iValue, iValue * iValue);

  return;
}

/**
 * Function Name: printSumTwoInt()
 * Description:   Computing and displaying the sum of
                  two integers
 * Pre:           Nothing
 * Post:          None
 */
void printSumTwoInt() {
  int i1;
  int i2;
  printf("\n  Computing sum of two int's -- printSumTwoInt():\n"\
    "\tEnter an integer + ENTER: ");
  scanf("%d", &i1);

  printf("\tEnter an integer + ENTER: ");
  scanf("%d", &i2);

  printf("\n\tThe sum of %d and %d is %d\n", i1, i2, i1 + i2);

  return;
}

/**
 * Function Name: printClassInfo()
 * Description:   Printing the class information
 * Pre:           Nothing (nothing is sent to this function)
 * Post:          None
 */
void printClassInfo() {
  printf("\n\tCIS 6 : Introduction to Programming"\
    "\n\tLaney College\n");

  return;
}

/** PROGRAM OUTPUT

        CIS 6 : Introduction to Programming
        Laney College

  Computing square of int -- printSquare():
        Enter an integer + ENTER: 5

        The square of 5 is 25
```

```
  Computing sum of two int's -- printSumTwoInt():
        Enter an integer + ENTER: 4
        Enter an integer + ENTER: 5

        The sum of 4 and 5 is 9
*/
```

Example 2

```
/**
 * Program Name: cis6L0512.c
 * Discussion:   Functions with arguments
 */
#include <stdio.h>

struct Fraction {
  int num;
  int denom;
};

struct FractionList {
  struct Fraction* frPtr;
  struct FractionList* next;
};

struct FractionNode {
  struct Fraction* frDataPtr;
  struct FractionNode* next;
};
/*Function prototypes*/

int insertFirst(struct FractionNode** listAddr,
                struct FractionNode* frDataPtr);

/*Application driver*/
int main() {

  insertFirst(0, 0); // BAD Testing!

  return 0;
}

/* Function definitions*/

/**
 * Function Name: insertFirst(struct FractionNode** listAddr,
                              struct FractionNode* frDataPtr);
 * Description:   Inserting node at the front of linked list
 *
 * Pre/Entry:     listAddr : Addres of the list
 *                frDataPtr : Address of the data (some given Fraction)
 *
 * Internals:     The given linked list is updated
 *
```

```
 * Post/Exit:     Returns a Boolean value
 */
int insertFirst(struct FractionNode** listAddr,
                struct FractionNode* myNodeAddr) {
  if (myNodeAddr != NULL) {
    if (*listAddr != NULL)
    myNodeAddr->next = *listAddr;

    *listAddr = myNodeAddr;

    return 1;
  } else {
    return 0;
  }
}
```