PROGRAMMING METHODOLOGY

Lab 3: Review 1 / Function and Program Structures

1 Introduction

In this lab tutorial, we focus on:

- Reviewing conditional and loop structures in Exercises section.
- Function and program structures.

2 Function and Program Structures

Functions[1] break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes.

C has been designed to make functions efficient and easy to use; C programs generally consist of many small functions rather than a few big ones. A program may reside in one or more source files. Source files may be compiled separately and loaded together, along with previously compiled functions from libraries. We will not go into that process here, however, since the details vary from system to system.

A function is often executed (called) several times, from several different places, during a single execution of the program. After finishing a subroutine, the program will branch back (return) to the point after the call.

As a basic example, suppose you are writing code to print out the first 5 squares of numbers, do some intermediate processing, then print the first 5 squares again. We could write it like this:

[1] Other programming languages may distinguish between a "function", "subroutine", "subprogram", "procedure", or "method"

```
1     // function1.c
2     #include <stdio.h>
3
4     int main()
5     {
6           int i;
7
8           // First time
9           for(i = 1; i <= 5; i++)
10          {
11                printf("%d\n", i * i);
12          }
13
14          // Second time
15          for(i = 1; i <= 5; i++)
16          {
17                printf("%d\n", i * i);
18          }
19
20          return 0;
21    }
```

Figure 1 Sample program

In **Figure 1**, we can easily see that there is a duplicate of the same procedure (in line 9 and 15). Thus, we may want to somehow put this code in a separate place and simply jump to this code when we want to use it.

```
1   // function2.c
2   #include <stdio.h>
3
4   void printSquare();
5
6   int main()
7   {
8         int i;
9
10        printSquare();
11        printSquare();
12
13        return 0;
14  }
15
16  void printSquare()
17  {
18        int i;
19        for(i = 1; i <= 5; i++)
20        {
21              printf("%d\n", i * i);
22        }
23  }
```

A function is like a black box. It takes in input, does something with it, then spits out an answer. Note that a function may not take any inputs at all, or it may not return anything at all. In the above example, if we were to make a function of that loop, we may not need any inputs, and we aren't

returning anything at all (Text output doesn't count - when we speak of returning we mean to say meaningful data that the program can use).

We have some terminology to refer to functions:

- A function, call it *f*, that uses another function *g*, is said to *call g*. For example, *f* calls *g* to print the squares of ten numbers.
- A function's inputs are known as its ***arguments***.
- A function *g* that gives some kind of answer back to *f* is said to ***return*** that answer. For example, *g* returns the sum of its arguments.

In runtime, or execution time, when the program pointer references to ***return*** statement of function, it will exit the current program, that means, the statements behind that ***return*** will not be executed.

2.1 Writing function in C

In general, if we want to declare a function, its format is as follows:

```
return-type function-name (type1 arg1, type2 arg2, …)
{
      declarations and statements
      return statement
}
```

We've previously said that a function can take no arguments, or can return nothing, or both. To do this, we use C's ***void*** keyword, as in **Figure 1**. Let's see the following example, a function takes one parameter and returns an integer number.

```
1   // function3.c
2   #include <stdio.h>
3
4   // Global decleration
5   int isEven(int);
6
7   int main()
8   {
9         int number;
10
11        printf("Input number = ");
12        scanf("%d", &number);
13
14        printf("%d", isEven(number));
15
16        return 0;
17  }
18
19  // Function definition
20  int isEven(int n)
21  {
22        if(n % 2 == 0)
23              return 1;     // n is even number
24
25        return 0;
26  }
```

In the above example, we declare a function, named ***isEven***, which check whether a number is even or not. In line 5, we define a function signature[2] of ***isEven***, without the implementation. Then, in line 20, we implement the ***isEven*** function. The function receives one input and returns {1, 0}, which **1** indicates the input number is even, and vice versa.

2.2 Program structure

In C, your program is controlled within ***main*** function, and all source code in ***main*** is called the body of the main function. The ***main*** function can be preceded by documentation, preprocessor statements and global declarations. The following figure shows the basic structure of C program.

[2] https://developer.mozilla.org/en-US/docs/Glossary/Signature/Function

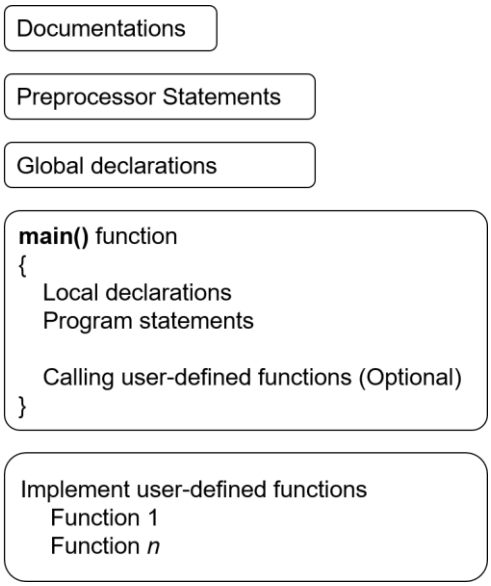

Figure 2 Basic structure of C program

To illustrate each part in **Figure 2**, let's see the below program.

```
1   // Filename: function-sample.c
2
3   // Preprocessor statements
4   #include <stdio.h>
5   #include <math.h>
6   #define PI 3.14159
7
8   // Global decleration
9   double getArea(double);
10
11  int main()
12  {
13          double radius;
14
15          do
16          {
17                  printf("Radius = ");
18                  scanf("%lf", &radius);
19          } while(radius < 0);
20
21          printf("Area  = %lf", getArea(radius));
22
23          return 0;
24  }
25
26  // Function definition
27  double getArea(double r)
28  {
29          if(r < 0)
30                  return -1;
31          return 2.0 * PI * pow(r,2);
32  }
```

In line `6` of the above program, we define a macro, named `PI`, to hold the value of π. Object-like macros were conventionally used as part of good programming practice to create symbolic names for constants, instead of hard-coding the numbers throughout the code. An alternative in both C and C++, especially in situations in which a pointer to the number is required, is to apply the ***const*** qualifier to a global variable. This causes the value to be stored in memory, instead of being substituted by the preprocessor. In line `31`, in order to compute x to the power of y, we include the pre-defined ***math.h*** library.

3 Exercises

1. Write a C function to print sum of all even numbers between 1 to n. Validating the input, in case the input isn't correct, prompt user to enter it again.
2. Write a C function to print sum of all even numbers between 1 to n. Validating the input, in case the input isn't correct, prompt user to enter it again.
3. Write a C function to print table of any number.
4. Write a C function to enter any number and calculate sum of all natural numbers between 1 to n. Validating the input, in case the input isn't correct, prompt user to enter it again.
5. Write a C function to find first and last digits of any number.
6. Write a C function to calculate sum of digits of any number.
7. Write a C function to calculate product of digits of any number.
8. Write a C function to count number of digits in any number.
9. Write a C function to swap first and last digits of any number.
10. Write a C function to print an input number in reverse order.
11. Write a C function to enter any number and check whether the number is palindrome or not.
12. Write a C function to check whether a number is Prime number or not. Validating the input, in case the input isn't correct, prompt user to enter it again.
13. Write a C function to check whether a number is Armstrong number or not.

14. Write a C function to check whether a number is Perfect number or not.
15. Write a C function to print all Prime numbers between 1 to *n*. Validating the input, in case the input isn't correct, prompt user to enter it again.
16. Write a C function to print all Armstrong numbers between 1 to *n*. Validating the input, in case the input isn't correct, prompt user to enter it again.
17. Write a C function to print all Perfect numbers between 1 to *n*. Validating the input, in case the input isn't correct, prompt user to enter it again.
18. Write a C function to convert Decimal to Binary number system.
19. Write a C function to compute the Factorial of *n*. Validating the input, in case the input isn't correct, prompt user to enter it again.

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n. \ (n \geq 0)$$

4 Reference

[1] Brian W. Kernighan & Dennis Ritchie (1988). *C Programming Language, 2nd Edition*. Prentice Hall.
[2] Paul Deitel & Harvey Deitel (2008). *C: How to Program, 7th Edition*. Prentice Hall.
[3] *C Programming Tutorial* (2014). Tutorials Point.
[4] *C Programming* (2013). Wikibooks.