COMP2006 Computer Organization
Functions

Functions allow you to modularize a program. All variables defined in function definitions are local variables, they can be accessed only in the function in which they are defined. Most functions have a list of parameters that provide the means for communicating information between functions via arguments in function calls. A function's parameters are also local variables of that function.

For example, we are going to create a *Factorial Calculator* using C language that accepts the user input and then outputs the answer.

We know that:

```
0! = 1
1! = 1
5! = 5x4x3x2x1
10! = 10x9x8x7x6x5x4x3x2x1
```

Then, we may write a C program as follows:

```c
#include <stdio.h>

int main() {
   long n, result, i;
   printf("\n\nFractorial Calculator\n");
   printf("======================\n\n");
   while(1) {
      printf("Please input an integer in the range from 0 to 25\n");
      scanf("%ld", &n);

      if (n <= 1)
         result = 1;
      else {
         result = n;
         for(i=n-1; i>1; i--)
             result = result * i;
      }

      printf("%ld! = %ld\n\n", n, result);
   }
   return 0;
}
```

However, it is not a good implementation because we put everything in the **main** function. Imagine that we work on a project that the program consists of more than a thousand lines of code. We cannot put everything in a single function because it is not easy to read and troubleshoot. In addition, many redundant lines of code are included in the program.

With the best implementation, we should modularize the program code by defining different functions for different specific tasks.

Function Definitions

To define a function, we need to provide the function name, return type and parameters. The following is the format of a function definition:

```
return_type function_name(parameter-list) {
   statements
}
```

For example, we define a function to calculate the factorial:

```
long factorial(long n) {

   long result, i;

   if (n == 1 || n == 0)
      return 1;

   result = n;

   for(i=n-1; i>1; i--)
      result  = result * i;

   return result;
}
```

The function name is **factorial**. It accepts a parameter n in long data type. The return type of the function is long.

Note: Once the **return** statement is executed, the remaining statements in the function will not be executed.

After defining the function, we can call it as follows:

```
int main() {
   long n, result;

   printf("\n\nFractorial Calculator\n");
   printf("=====================\n\n");
   while(1) {
      printf("Please input an integer in the range from 0 to 20\n");
      scanf("%ld", &n);

      result = factorial(n);

      printf("%ld! = %ld\n\n", n, result);
   }

   return 0;
}
```

Now, the **main** function handles the console inputs and outputs. The **factorial** function performs the calculation.

Function Prototypes

The function prototypes (declarations) tells the compiler about the number of parameters the function takes, the data-types of the parameters, and the return type of the function. By using this information, the compiler cross-checks the function calls.

In C program, we must declare a function before we can call it – the function declaration must be put above the caller. Consider the following example code:

```
#include <stdio.h>

int main() {
   saySomething();

   return 0;
}

void saySomething() {
   printf("Just to say Hi!\n");
}
```

You can see that the **saySomething** function is put after its caller, the **main** function. Some compilers detect this problem and prompt the compile errors. Without the compiler detecting the errors, the function calls could result in fatal runtime errors or cause subtle, difficult-to-detect problems.

To fix the problem, we can either (1) the saySomething function before the main function, or (2) add the function prototype of the saySomething before the main function. The second method is preferred for the large-scale programs with many functions and the functions may call other functions.

```
#include <stdio.h>

void saySomething();  // function prototype

int main() {
   saySomething();

   return 0;
}

void saySomething() { // function prototype
   printf("Just to say Hi!\n");
}
```

We put the function prototypes at the beginning of the code file (after the #include statements). Then, we can put the function definition anywhere we want.

Call by Value

By default, a parameter will be passed to the function by value – a copy of the value is passed to the function. Any changes to the parameter inside the function will not affect the original value of the parameter. Consider the following example code:

```c
#include <stdio.h>

void doSomething(int n) {
   n = n * 10;
   printf("doSomething: %d\n", n); // doSomething: 100
}

int main() {
   int x = 10;
   doSomething(x);
   printf("main: %d\n", x);        // main: 10

   return 0;
}
```

The integer **x** is passed to the **doSomething** function. In the **doSomething** function, the parameter **n** is updated. But, after the execution of the function, the value of the integer **x** is not changed.

The following is another example with function:

```c
#include <stdio.h>

int power(int n, int m) {
    int i;
    int t = n;

    if (m == 0)
        return 1;

    for (i = 0; i<m; i++)
        t *= n;

    return t;
}

int main() {
    int a = 2;
    int b = 8;
    int c = power(a, b);
    printf("%d^%d = %d\n", a, b, c);
    return 0;
}
```

We pass two values to the function and it returns a value to the caller.

Call by Reference

Also consider the following example code:

```c
#include <stdio.h>

void doSomething(int *n) {
    *n = *n * 10;
    printf("doSomething: %d\n", *n);   // doSomething: 100
}

int main() {
    int x = 10;
    doSomething(&x);
    printf("main: %d\n", x);           // main: 100

    return 0;
}
```

In the declaration of the **doSomething** function, the parameter **n** is defined with the *asterisk sign (*)* that is called “pointer”. The pointer is a variable, but it is not used for storing the value. It stores a memory address instead.

On the other hand, the **doSomething** function is called with the parameter **&x**. The *ampersand sign (&)* represents the memory address (reference) of the variable. The method of calling a function with the reference of the variables is named “call by reference”.

Because the program knows the memory address of the variable, it can write the value to the memory address directly and overwrite the original value.

Application of Call by Reference

Consider the following example:

```c
#include <stdio.h>

void swap(int *x, int *y) {
    int z = *x;
    *x = *y;
    *y = z;
}

int main() {
    int a = 2;
    int b = 8;
    printf("before swapping: a = %d & b = %d\n", a, b);
    swap(&a, &b);
    printf("after swapping: a = %d & b = %d\n", a, b);
    return 0;
}
```

In C language and most programming language, a function can return one value. But we sometimes may want to get two output values or more from the function. For this situation, we can use call-by-reference approach because it can modify the values of input parameters.

In the example above, the **swap** function accepts two input parameters (pointers, reference addresses of the variables). Then, it swaps the values of them.

Exercise 1

Write a C program that has two functions – **main** and **maximum**.

- The **maximum** function accepts three integers and returns the maximum one.
- The **main** function prompts the user to input three integers, passes these three integers to the **maximum** function, and prints the result of the **maximum** function.

The following is the sample input and output:

```
Enter three integers: 27 93 42
The maximum is: 93
```

Name your C code file to **exe1.c** and submit it to Moodle.

Exercise 2

Write a C program that has two functions – **main** and **sort**.

- The **sort** function accepts three integers and sort them in ascending order and put the result back to the original variables. Note that the return type of the **sort** function is void.
- The **main** function prompts the user to input three integers, passes these three integers to the **sort** function, and prints the sorted results after invoking the **sort** function.

Name your C code file to **exe2.c** and submit it to Moodle.