Tutorial 6 : 6 September

1 Why use functions?

Modular programming[1] is a software design technique that emphasizes separating the program into independent, inter-changeable modules (sections), such that each module (section) contains everything necessary to execute only one aspect of the desired task. In C, modular programming mainly means using functions to break your program into smaller and simpler modules.

The following are the advantages for using functions [2] [3]:

- **Reusability :** Once a function is defined, it can be used over and over and over again. You can invoke the same function many times in your program, which saves you work. Imagine what programming would be like if you had to teach the computer about sines every time you needed to find the sine of an angle! You'd never get your program finished!
- **Abstraction :** In order to use a particular function, you don't have to know how it works inside! All you need to understand is how to interact with the *interface* of the function, i.e. understand how to call the function. It is sort of like driving a car or using a telephone. With an automobile, you don't need to understand every detail about the engine and gears and wheels, if all you want to do is drive the car. Take your favorite function printf() (Yes, it is a function) for example; you've been using it all along even though you do not know how it makes the computer print stuff.
- **Testing :** Because functions reduce code redundancy, theres less code to test in the first place. Also because functions are self-contained, once we have tested a function to ensure it works, we do not need to test it again unless we change it. This reduces the amount of code we have to test at one time, making it much easier to find bugs (or avoid them in the first place).

2 The main() function

This section explains how **main() itself is a function**.

You read it correctly, since your very first program in C you have been using functions but without knowing about them till recently. In C, the *main()* function is treated the same as every other function, it has a return type (and in some cases accepts inputs via parameters). The only difference is that the main function is "called" by the operating system when the user runs the program. Thus execution of your program always begins with the main() function. Try running a C code without main() function and note the error.
Since main() is just like any other function, it can have any return type like float or void(explained later) and not just int, but are generally not preferred.

3 Find the output for all the following problems

3.1 Problem 1

```
1  #include <stdio.h>
2
3  void ping(int moveCount) {
4      printf("Move %d : Ping\n", moveCount);
5      return;
6  }
7
8  int main() {
9      int moveCount = 1;
10     ping(moveCount);
11     return 0;
12 }
```

Output :
Move 1 : Ping

Comments :

- Note that the function *ping()* in above code has return type *void*. Void is the type for the result of a function that does not provide a return value to its caller. Usually such functions are called for their **side effects** (Changes caused by calling the function), such as writing to the output.
  A function with void return type gets completed either by reaching the end of the function or by hitting a return statement. Therefore the return statement is optional in above code.

  Go through the following code and try to understand whether the return statements are optional or not i.e. Would the function perform the same task in case we remove one of the return statements?

```
1  void isPrime(int n) { // Assume n >= 2
2      int i;
3      for(i=2; i<=n/2; ++i)
4      {
5          if(n%i==0)
6          {
7              printf("Not prime\n");
8              return;
9          }
10     }
11     printf("Prime\n");
12     return;
13 }
```

3.2 Problem 2

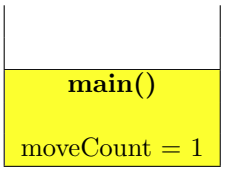
```
1  #include <stdio.h>
2
3  int pong(int moveCount);
4
5  int ping(int moveCount) {
6    printf("Move %d : Ping\n", moveCount);
7    moveCount = pong(moveCount = moveCount + 1);
8    return moveCount;
9  }
10
11 int pong(int moveCount) {
12   printf("Move %d : Pong\n", moveCount);
13   return moveCount+1;
14 }
15
16 int main() {
17   int moveCount = 1;
18   moveCount = ping(moveCount);
19   printf("moveCount = %d\n", moveCount);
20   return 0;
21 }
```

Output :
Move 1 : Ping
Move 2 : Pong
moveCount = 3

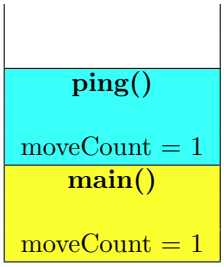
Explanation :
To understand this problem, we need to understand the *stack* maintained by the system to perform function calls. For that, we would recommend you to once go through the following short video tutorial before reading any further: https://www.youtube.com/watch?v=Q2sFmqvpBe0

Even though the tutorial can be a bit advanced for this course, what is important to understand from it is that the local variables for each function call exists on the corresponding stack frame only. Therefore once the function execution completes, these variables are removed from the memory along with the corresponding stack frame.
Now we will try to understand the above program in terms of the call stack. A stack can be thought of as a stack of books, where you can only put and take off things from the top.

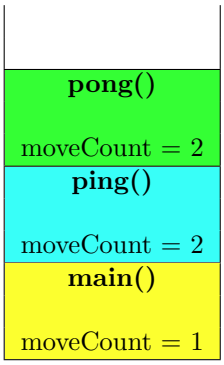
- When the program execution begins, imagine only one 'stack-frame' (The variables for the specific function call) on the top of stack which corresponds to the main() function. Therefore just before executing statement 18, the stack will be:



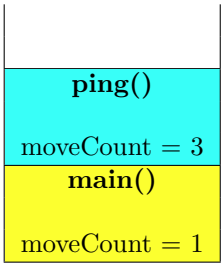
- Now when main() calls ping() in line 18, another stack frame can be imagined to be pushed on the stack for the function ping() with the passed parameter values. It is important to remember that both of these moveCount variables are different and should not be confused with each other. Just before executing statement 6, the stack can be imagined as:



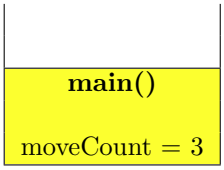
- When executing instruction 7, the arguments of function call are computed first before calling the function. Thus the value of moveCount becomes 2 and that value is passed to the function pong() (Remember that a = b returns the value of b). Therefore just before executing instruction 12, the stack can be imagined as the following:



- Now when the function call for pong() completes, the value 3 is returned to the calling function ping() whose stack frame is just below it. So now the stack frame of function pong() will be removed from the top and the execution returns to statement 7 in function ping(). Note that the value of moveCount changes. Now, just before executing statement 8, the stack can be imagined as the following :



- Now when the function call for ping() completes, the value 3 is returned to the calling function main() whose stack frame is just below it. So now the stack frame of function ping() will be removed from the top and the execution returns to statement 18 in main(). Therefore just before executing statement 19, the stack can be imagined as the following :



To try :
Find the output for the above code if statement 18 is replaced by

```
moveCount = ping(pong(ping(moveCount)));
```

Hint : Just remember that function arguments are computed before calling the function.

3.3 Problem 3

```
1  #include <stdio.h>
2
3  void pong() {
4    static moveCount = 1;
5    printf("Move %d : Pong\n", moveCount);
6    moveCount++;
7    return;
8  }
9
10 void ping() {
11   static moveCount = 1;
12   printf("Move %d : Ping\n", moveCount);
13   moveCount++;
14   pong();
15   return;
16 }
17
18 int main() {
19   ping();
20   pong();
21   ping();
22   return 0;
23 }
```

Output :
Move 1 : Ping
Move 1 : Pong
Move 2 : Pong
Move 2 : Ping
Move 3 : Pong

Comments :

- Since static variables are not defined on the stack, static variables continue to exist even after the block in which they are defined terminates. Thus, the value of a static variable in a function is retained between repeated function calls to the same function.
- The scope of static variables is confined to the block in which it is defined; however, the memory allocated to such variables becomes permanent for the duration of the program.

Use the above ideas to understand the output generated by the program.

4 Practice problems

4.1 Find output

```c
#include <stdio.h>

int moveCount = 1;

void pong() {
  printf("Move %d : Pong\n", moveCount);
  moveCount++;
  return;
}

void ping() {
  printf("Move %d : Ping\n", moveCount);
  moveCount++;
  pong();
  return;
}

int main() {
  ping();
  pong();
  ping();
  return 0;
}
```

4.2 Find output

```c
#include <stdio.h>
int x = 1;
int main() {
  x = 3;
  {
    int x = 7;
    {
      x = 2;
    }
    printf("%d \n", x);
  }
  printf("%d \n" ,x);
  return 0;
}
```

4.3 Find output (Also decipher what it does)

```c
#include <stdio.h>
#define R 3
#define C 6

void doSomething(int m, int n, int a[][C])
{
    int i, k = 0, l = 0;

    while ((k<m) && (l<n)) {
        for (i = l; i < n; ++i)
            printf("%d ", a[k][i]);
        k++;

        for (i = k; i < m; ++i)
            printf("%d ", a[i][n-1]);
        n--;

        if (k<m) {
            for (i = n-1; i >= l; --i)
                printf("%d ", a[m-1][i]);
            m--;
```

```
        }

        if (l<n) {
            for (i = m-1; i >= k; --i)
                printf("%d ", a[i][l]);
            l++;
        }
    }
}

/* Driver program to call the above function */
int main() {
    int a[R][C] = { {1,  2,  3,  4,  5,  6},
        {7,  8,  9,  10, 11, 12},
        {13, 14, 15, 16, 17, 18}
    };
    doSomething(R, C, a);
    return 0;
}
```

4.4 Find output (Also understand what it does

```
#include <stdio.h>

long value(int);
void foo(int [], int);
float bar(int [], int);

int main()
{
  int n = 5;
  int a [6];
  foo(a, n);
  printf ("%.2f \n", bar(a, n));
  return 0;
}

void foo(int a[], int n) {
  int i, j;
  int b[6];
  for(i = 0; i < n; i++) {
    for(j = 0; j <= ( n - i - 2 ); j++)
      printf(" ");

    for (j = 0; j <= i; j++) {
      b[j] = value(i) / (value(j) * value(i - j)) ;
      printf("%d ", b[j]);
    }
    printf("\n");
    a[i] = (int)bar(b, i + 1);
  }
}

float bar (int a[], int n) {
  if (n%2 == 0)
    return (a[n/2] + a[n/2 -1]) / 2.0;
  return (float) a[n/2] ;
}

long value (int n) {
  int c;
  long result = 1;
  for (c = 1; c <= n; c++)
    result = result * c;
  return result ;
}
```

5 Answers for practice problems

5.1 Problem 1

Move 1 : Ping
Move 2 : Pong
Move 3 : Pong
Move 4 : Ping
Move 5 : Pong

5.2 Problem 2

2
3

5.3 Problem 3

1 2 3 4 5 6 12 18 17 16 15 14 13 7 8 9 10 11

5.4 Problem 4

```
                    1
                1       1
            1       2       1
        1       3       3       1
    1       4       6       4       1
2.00
```

References

[1] Wikipedia. Modular programming — Wikipedia, the free encyclopedia.

[2] Eric N. Eide. Why use functions? https://www.cs.utah.edu/~zachary/computing/lessons/uces-10/uces-10/node11.html.

[3] Alex. Why functions are useful, and how to use them effectively. http://www.learncpp.com/cpp-tutorial/1-4b-why-functions-are-useful-and-how-to-use-them-effectively/.