To:
Tenouk

# C LAB WORKSHEET 13
# C & C++ Functions Part 4: With Return Values

1. Understanding and using C/C++ functions that receive and return value.
2. Tutorial references that should be used togeC++ function part 1, function part 2, function part 3 and function part 4.
3. For main() function complete story please refer to C and C++ main() story.

**Receiving And Returning Values From Functions**

In the previous exercises we saw how arguments are passed to a function. We learned the difference between passing a scalar (non-array) and passing an array. When passing a scalar, the calling function doesn't have to be concerned that the called function will change its value because the called function (callee) has received **a copy** of the original scalar.
Altering the copy doesn't change the original variable in the calling function. However, when passing an array, the array is stored in only one place and **no copy has been made**. The called function receives the address of the array (for example, if the entire array is passed to the function, the address of the array that has been passed is **a pointer to the first array element**). Now if the called function changed the array, then that change would reflect the contents of the array in the calling function. The term "**pass by reference**" normally used for this type of passing arguments to functions versus the "**pass by value**" if passing a scalar.
In this worksheet we will learn how values may be returned from the called function (callee) to the calling function (caller). You will see that only one value can be returned. Take note the term used for "passing to functions" and "returning.from functions". The return statement will be a new statement for us though it is just a normal return statement that you already found at the end of the main() function. With the return statement, we will be able to return values to the calling function upon the completion of the called function. Many items are different when writing a program with a function that returns a value. These items are outlined in the following sample code:

```
#include <stdio.h>

// this function receive nothing but
// return a char...
char JustTesting(void);

void main(void)
{
    char x;
    x = JustTesting();
    printf("The returned char is = %c\n", x);
}

char JustTesting(void)
{
    return ('T');
}
```


In the prototype as well as in the function header, we must state the **data type** of the item that the function returns. Here, we are returning a character. Then at the end of the function, it should return that type of value using the return statement. Finally, when calling the function, the calling function should do something with the item it will receive. Here, main() assigns the received character to the variable x. main() could have also called the function inside a printf(), for instance and print the received item directly. The calling function can perform other tasks with the received item. Calling the function JustTesting(); without assigning it to x or printing it would be wrong.
The following code samples summarizes the various ways of using functions. We can categorize functions by whether or not arguments are **passed to them**. Or we can categorize them by whether or not values are **received from them**. Intersecting these two methods of categorizing functions, we come up with four basic methods of writing and using functions. We will use some of the functions categories in our next program examples.

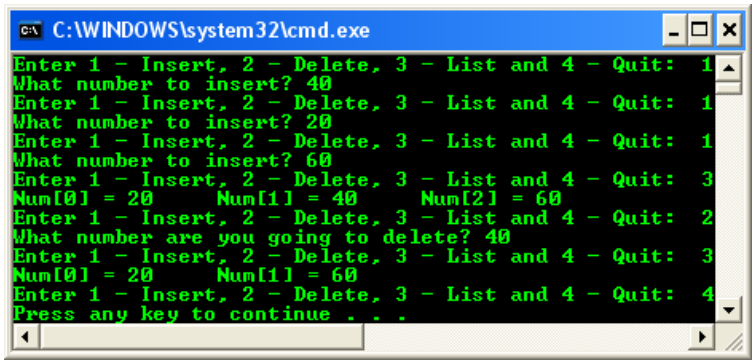
| | Does not pass argument | Does pass arguments |
|---|---|---|

| | | |
|---|---|---|
| **No return** | void main(void)<br>{<br>  TestFunct();<br>  ...<br>}<br><br>void TestFunct()<br>{<br>  // receive nothing<br>  // and nothing to be<br>  // returned<br>} | void main(void)<br>{<br>  TestFunct(123);<br>  ...<br>}<br><br>void TestFunct(int i)<br>{<br>  // receive something and<br>  // the received/passed<br>  // value just<br>  // used here. Nothing<br>  // to be returned.<br>} |
| **With a return** | void main(void)<br>{<br>  x = TestFunct();<br>  ...<br>}<br><br>int TestFunct(void)<br>{<br>  // received/passed<br>  // nothing but need to<br>  // return something<br>  return 123;<br>} | void main(void)<br>{<br>  x = TestFunct(123);<br>  ...<br>}<br><br>int TestFunct(int x)<br>{<br>  // received/passed something<br>  // and need to return something<br>  return (x + x);<br>} |

```
#include <stdio.h>

// prototypes, the parameter names are optional,
// print the menu and obtains a selection
int GetChoice();
// inserts a number in the sorted array.
int AddNum (int num[], int sz);
// removes a number from the array.
int DelNum(int num[], int sz);
// prints out the sorted array
void PrintNums(int num[], int sz);

void main()
{
    // the number of elements in array A[]
    int Size = 0,
    // the array that will keep its numbers sorted
    A[20],
    // the selection made from the menu
    Selection;

    // keep doing this loop until a 4 or Quit is
    // selected from the menu...
    Selection = GetChoice();
    for( ; Selection != 4; )
    {
        if(Selection == 1)
            Size = AddNum(A, Size);
        else
            if(Selection == 2)
                Size = DelNum(A, Size);
            else
                PrintNums(A, Size);
        Selection = GetChoice();
    }
}

// assuming that user will enter only a valid int
int GetChoice()
{
    int Choice;

    printf("Enter 1 - Insert, 2 - Delete, ");
    printf("3 - List and 4 - Quit:  ");
        scanf_s("%d", &Choice);
        return Choice;
}

int AddNum(int Num[ ], int sz)
{
    // local variables...
    int i, j, Number;
    // gets the number to insert
```

```
    printf("What number to insert? ");
    scanf_s("%d", &Number);
    // finds the place (i) to put the new number
    for(i = 0; i < sz; ++i)
        if(Number < Num[i])
            break;
    // shift the rest of the array by moving
    // the numbers up by one slot.
    for(j = sz; j > i; --j)
        Num[j] = Num[j - 1];
    // place the new number
    Num[i] = Number;
    // the array size is incremented
    // and return to the calling function
    return ++sz;
}

// assuming that the deleted item exists
int DelNum(int Num[], int sz)
{
    // local variables...
    int i, Number;
    // gets the number to be deleted
    printf("What number are you going to delete? ");
    scanf_s("%d", &Number);
    // find the place in the array to be deleted
    for(i = 0; i < sz; ++i)
        if(Number == Num[i])
            break;
    // the array size is decremented
    --sz;
    // shift the rest of the array by moving the numbers
    // down by one slot
    for( ; i < sz; ++i)
        Num[i] = Num[i + 1];
    // return sz to the calling function
    return sz;
}

void PrintNums(int Num[ ], int sz)
{
    // local variable...
    int i;

    for(i = 0; i < sz; ++i)
        printf("Num[%d] = %d\t", i, Num[i]);
    printf("\n");
}
```

---




Looking at the output, you can see that a menu is presented again and again until the user enters the number 4, at which time the program terminates. The user has three other options. If a 1 is entered at the menu, then a number provided by the user is inserted in a list of existing numbers. This list of number is kept in ascending order. If a 2 is entered, then a number is deleted and if a 3 is entered, then the ordered list of numbers is printed for the user.

In the output we see that the user inserts 40, 20 and 60 in that order, in the list. When asked to print the list, the program prints them in order. Then the user asks that 40 be removed from the list. After printing the list, we see that 40 is removed. only after examining the Num[ ] array's contents, we can see how the list is maintained. Notice that this is not a linked list.

In main(), Size stores the number of elements in the list. Initially, this number is zero. We also have a 20-element integer array called A[ ]. Firstly main() calls the GetChoice() function. Nothing is passed to this function. It prints the menu with four choices, reads the menu option from the user and returns that value. This value is then stored in the variable Selection in main(). This process of obtaining the value for Selection is repeated at the bottom of the for loop until an option of 4 or Quit is selected.

Inside the loop, if Selection is equal to 1, then the function AddNum() is called. When calling this function, the array and the size of the array are passed as arguments. The function returns the new size of the array which is then assigned to Size in main().

If A[ ] and Size were defined, not inside main(), but outside it, as the prototype are declared, then we would not have to send them as arguments or need to receive Size. These variables would be accessible to all function and they would be called **global**, not **local variables**. Global variables should be avoided because any function may have access to them and may be inadvertently altered by some function in the program. The larger the program, the more likely this is.

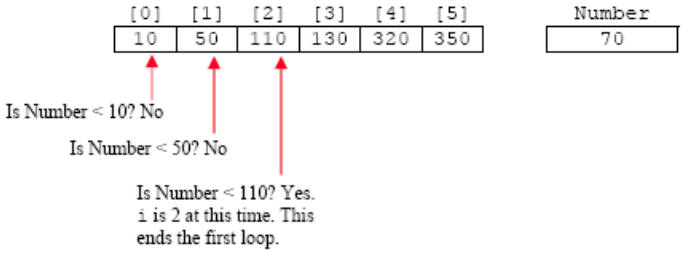
AddNum() doesn't check to see if the insertion was successful or not. If it did, then it could return the same Size as it received. This enhancement can be added later. For now, two equal numbers may coexist in the list.

DelNum() is a function similar to AddNum(). However it also doesn't check to see if the deletion was successful or not. If a user wants to delete a number that doesn't exist in the list, the function would not give an error message and Size would be decremented anyway. This fine tuning can also be added later. The function are kept simple since this is your fist example.

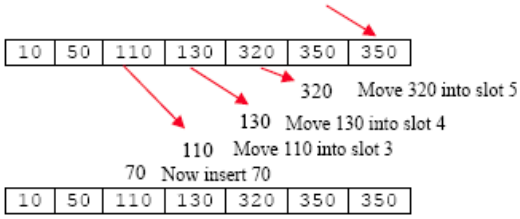
The PrintNums() function receives the array Num[ ] and its size, sz. The for loop goes through all

the slots of the array and prints the list of numbers. The numbers are already in order when control of execution comes to this function.
The AddNum() function first receives the number from the user and it must be placed in the array. The function uses two loops to insert that number. The first loop looks to find the location where that number belongs in the list and the second loop shifts to the right all the elements from that point by one slot, making space for new number. After this loop, the number is inserted using one assignment statement. After incrementing sz, the new size is returned.

From the following Figure, sz is equal to 6 because there are 6 elements in the array. Number is 70. It is the number that we have to insert in the proper place. Starting from the left, we see if 70 is less than 10 and if 70 is less than 50 and finally we come to the point where 70 is less than 110. In the function, i is equal to 2 and we use a break to exit the loop. We now know that the 70 belongs in slot number 2.
Then the second or the j loop starts from the other end of the array, sliding its elements by one slot to the right. The 350 is moved to slot 6, the 320 to slot 5 and so on until the 110 is moved to slot 3. Now there is room for 70 in slot 2, so it is inserted. Notice that the j loop terminates when j becomes i. The following Figure shows the procedure to insert 70 in the ordered list.




Second loop starts with j equal to sz or 6. Move 350 into slot 6.

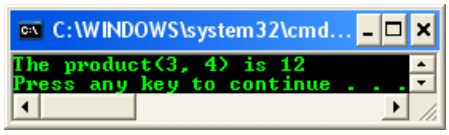



The DelNum() function works similarly. The first loop merely looks for the number to be deleted. When it is found, its location is preserved in i by a break. sz is decremented. Instead of shifting the elements of the array to the right, the second loop shifts the elements to the left, writing over the element to be deleted. The new size is returned. Remember that in order to complete writing this function, it should handle instances where the number to be deleted doesn't exist in the array in the first place.

**Recursion: Recursive Function**

When a function calls itself, it is called recursion. So far we just deal with iterations in loops but recursion allows us to reduce a large amount of complex code to a program that is smaller in length. Also, once recursion is mastered, the logic of it is much more simplified. LISP is a programming language that uses recursion extensively. The following program example uses recursive function, build and run the program.

```
#include <stdio.h>

int Product(int a, int b);
int Sum(int q, int r);

void main(void)
{
    int x;

    x = Product(3, 4);
    printf("The product(3, 4) is %d\n", x);
}

// only adding and subtracting,
// multiplication is done
int Product(int a, int b)
{
    if(a == 1)
        return b;
    else
        // calls itself, Product() calls Products()...
        return (Sum(b, Product(a - 1, b)));
}

// adding and subtracting by 1,
// 'q' and 'r' are added
int Sum(int q, int r)
{
    if(q == 0)
        return r;
    else
        // Sum() calls Sum()
        return(Sum(q-1, r) + 1);
}
```




This example uses two recursive functions, one is called Product() and the other is Sum(). We can easily tell that they are recursive functions because they call themselves. Within the definition of Product(), there is a call to Product() again. Also Sum() calls itself. First let us see how Sum() works, then we will look at Product(). This example assumes that our computer can only add or subtract the number 1. Using these two basic operations, the functions can be used to add or multiply two numbers. This example just to illustrate recursion and not to do something useful.

**Practices**

1. Build, run and show the output for the following program. Answer the questions that follows.

```
#include <stdio.h>

void Average(int, int);         // Line 1
void Message(void);             // Line 2

void main(void)                 // Line 3
{
  Message();                    // Line 4
  Average(30, 20);              // Line 5
}

void Average(int x, int y)      // Line 6
{
  // all the work is done in the function body
  printf("(x + y)/2.0 = %.2f\n", (x+y)/2.0);      // Line 7
}

void Message(void)              // Line 8
{
  // all work done here
  printf("The average is = ");  // Line 9
}
```

   a. Which one is a calling function (caller) and which one is a called function (callee)?
   b. Give the line number for each of the following descriptions.

   _____ - The beginning of the definition for the Message() function.
   _____ - The beginning of the definition for the main() function.
   _____ - The beginning of the definition for the Average() function.
   _____ - The prototype for Message().
   _____ - The prototype for Average().
   _____ - Where main() calls Message().
   _____ - Where Message() calls printf().
   _____ - Where main() calls Average().
   _____ - Where Average() calls printf().

   c. There are three user-defined functions. Name them.
   d. Which function (or functions) does not return anything? How can you tell that?
   e. Which function (or functions) does not receive any arguments? How can you tell that?
   f. Which function (or functions) receives arguments? How many?
   g. If the four lines used to define Message() are inserted between main() and Average(), would there be any problem running the code?

---

```
The average is = (x + y)/2.0 = 25.00
Press any key to continue . . .
```

   a. Caller is main(), Message() and Average() and callee are Message(), Average() and printf().
   b. Line numbers are:

   Line 8 - The beginning of the definition for the Message() function.
   Line 3 - The beginning of the definition for the main() function.
   Line 6 - The beginning of the definition for the Average() function.
   Line 2 - The prototype for Message().
   Line 1 - The prototype for Average().
   Line 4 - Where main() calls Message().
   Line 9 - Where Message() calls printf().
   Line 5 - Where main() calls Average().
   Line 7 - Where Average() calls printf().

   c. main(), Average() and Message().
   d. main(), Average() and Message(). This is based on the void keyword used at the beginning of their header.
   e. main() and Message(). This is based on the void keyword used in the parameter list.
   f. Average() receives two arguments.
   g. No problem because the definition still after its prototype declaration.

2. Next, let us have two Average() functions, one that returns a value and one that doesn't. Show the output and answer the questions.

```
#include <stdio.h>

void Average1(int, int);    // Line 1
float Average2(int, int);   // Line 2

void main(void)             // Line 3
{
  float Avg;                // Line 4
  Average1(30, 20);         // Line 5
  Avg = Average2(30, 40); // Line 6
  printf("Average = %.2f\n", Avg);    // Line 7
}

float Average2(int x, int y)    // Line 8
{
  return (float)((x + y) / 2.0);  // Line 9
}

void Average1(int x, int y)
{
  printf("(x + y) / 2.0 = %.2f\n", (x + y) / 2.0);
}
```

---

a. Which function returns a value? What in the prototype indicates that? What in the function definition indicates that?
b. Which function doesn't return a value? What in the prototype and the function definition indicates that?
c. Which function uses the keyword return?
d. Notice the difference between Line 5 and 6. Why isn't the function assigned to a variable when Average1() is called?
e. Why is the function assigned to a variable when Average2() is called?
f. Which function, Average1() or Average2(), prints the average?
g. Which function, Average1() or Average2(), returns the average instead of printing it?

```
(x + y) / 2.0 = 25.00
Average = 35.00
Press any key to continue . . .
```

a. Average2(). The float keyword at the beginning of the Average() prototype. The float keyword at the beginning of the Average() definition.
b. main() and Average1(). The void keyword at the beginning of their prototype and definition.
c. Average2().
d. Because Average1() does not return any value. Then, no value need to be assigned to any variable.
e. Because Average2() return a value of type float. The returned float will be assigned to variable Avg.
f. Average1() prints the average.
g. Average2() returns the average.