

# COP 3223: C Programming Spring 2009

## Functions In C – Part 3

Instructor : Dr. Mark Llewellyn  
markl@cs.ucf.edu  
HEC 236, 407-823-2790  
<http://www.cs.ucf.edu/courses/cop3223/spr2009/section1>

School of Electrical Engineering and Computer Science  
University of Central Florida



# Tracing Program Execution With Functions

- It is important in your understanding of how functions work, to be able to trace the values passed to a function when it is called, the execution effect of the function, and the value returned by the function when a return state is encountered.
- We'll trace the execution of the program shown on the next page; and I've put some additional tracing problems that involve functions in the practice problems. The code for each of these problems is also on the code page, however, before you download them and run them, trace the execution by hand and see if the execution verifies your hand trace.



```
4 #include <stdio.h>
5
6 int exampleFunction(int x, int y, int z)
7 {
8     int sum;
9     sum = x + y + z;
10    if (sum < x * y){
11        return x + y;
12    }//end if stmt
13    if (sum <= 2 * x * y){
14        return y + z;
15    }//end if stmt
16    return x + z;
17 }//end exampleFunction function
18
19 int main()
20 {
21     int a = 2, b = 3, c = 1;
22     c = exampleFunction(a + b, a + c, b + c);
23     printf("\nAfter first call to exampleFunction: a = %d b = %d c = %d\n\n", a, b, c);
24     b = exampleFunction(a, b, c);
25     printf("After second call to exampleFunction: a = %d b = %d c = %d\n\n", a, b, c);
26     a = exampleFunction( a, b, exampleFunction(c, b, a) );
27     printf("After third call to exampleFunction: a = %d b = %d c = %d\n\n", a, b, c);
28     printf("\n\n");
29     system("PAUSE");
30     return 0;|
```



## TRACE

Line 21: `a=2, b=3, c = 1`

Line 22: `c = exampleFunction(a + b, a + c, b + c)`, so calling `exampleFunction(5, 3, 4)`

Line 6: in `exampleFunction(x = 5, y = 3, z = 4)`

Line 9: `sum = x + y + z`, since  $5 + 3 + 4 = 12$ , `sum = 12`

Line 10: `if (sum < x * y)` is  $12 < 15$ , yes so Line 11 is executed next

Line 11: `return x + y` – function returns value of 8 – returns to Line 22

Line 22: `c = 8`

Line 23: prints: “After the first call to `exampleFunction`: `a = 2, b = 3, c = 8`”

Line 24: `b = exampleFunction(a, b, c)`, so calling `exampleFunction(2, 3, 8)`

Line 6: in `exampleFunction(x = 2, y = 3, z = 8)`

Line 9: `sum = x + y + z`, since  $2 + 3 + 8 = 13$ , `sum = 13`

Line 10: `if (sum < x * y)` is  $13 < 6$ , no so Line 13 is executed next

Line 13: `if (sum < 2 * x * y)` is  $13 < 12$ , no so Line 16 is executed next

Line 16: `return x + z` – function returns value of 10 - returns to line 24

Line 24: `b = 10`

Line 25: prints: “After the second call to `exampleFunction`: `a = 2, b = 10, c = 8`”



## TRACE

Line 26: `a = exampleFunction(a, b, exampleFunction(c, b, a))`

first call to `exampleFunction` is the inner call whose purpose is to return the value of the third parameter to the outer call to `exampleFunction`. The inner call is:

`exampleFunction(c, b, a)`, so the call is `exampleFunction(8, 10, 2)`

Line 6: in `exampleFunction(x = 8, y = 10, z = 2)`

Line 9: `sum = x + y + z`, since  $8 + 10 + 2 = 20$ , `sum = 20`

Line 10: `if (sum < x * y)`, is  $20 < 80$ , yes, so Line 11 is executed next

Line 11: `return x + y` – function returns value of 18 – returns to inner call in line 26

Line 26: `a = exampleFunction(2, 10, 18)`, since `a = 2`, `b = 10`, and inner call returned 18

Line 6: in `exampleFunction(x = 2, y = 10, z = 18)`

Line 9: `sum = x + y + z`, since  $2 + 10 + 18 = 30$ , `sum = 30`

Line 10: `if (sum < x * y)` is  $30 < 20$ , no, so Line 13 is executed next

Line 13: `if (sum < 2 * x * y)`, is  $30 < 40$ , yes, so Line 14 is executed next

Line 14: `return y + z` – function returns 28 – returns to line 26

Line 26: `a = 28`

Line 27: prints: “After the third call to `exampleFunction`: `a = 28`, `b = 10`, `c = 8`”



```
C:\Documents and Settings\HP_Administrator\Local Settings\Temporary I...  
After first call to exampleFunction: a = 2 b = 3 c = 8  
After second call to exampleFunction: a = 2 b = 10 c = 8  
After third call to exampleFunction: a = 28 b = 10 c = 8  
Press any key to continue . . . .
```

Did you get these values when you traced the execution of the program? If not...do it again!



# Passing Parameters In C

- Different programming languages employ several different mechanisms for passing parameters to functions.
- Two of the most popular mechanisms are **pass-by-value** and **pass-by-reference**. These two mechanisms are used in languages like Java, Perl, C, C++, C# and many others.
- **In C, parameter passing is done using pass-by-value**, as we have seen. An every C function returns at most 1 value.
- This of course means that it is not possible for a function in C to modify any of the actual parameters that are passed to it (see next page for example).



# Passing Parameters In C

- Recall that pass-by-value means that a copy of the value of the actual parameter is copied into the formal parameter in the function.
- Since the function is working with its own copy of each parameter, it is not possible for the function to modify the value of any parameter passed to it.

The called function

```
int aFunction(int a, int b, int c) { . . . }
```

The calling function

```
int x = 1;  
int y = 2, z = 3;  
result = aFunction( x, y, z );
```

Value of 1  
copied to a

Value of 2  
copied to b

Value of 3  
copied to c





# Passing Parameters In C

- Passing parameters by value places a serious restriction on what a function can accomplish.
- For example, suppose you wanted to construct a function that would take 3 parameters and add 10 to each parameter. Since the function can only return one value, this would not be possible to do in C with a single function call. (Rather you would write the function with a single parameter and then call it three different times. This is practice problem 2 in this set of notes).
- Fortunately, there is a way around this restriction in C which allows us to simulate the pass-by-reference mechanism. Pass-by-reference is simulated using pointers and the indirection operator.



# Passing Parameters In C

- Pass-by-reference differs from pass-by-value in that a “reference” to the parameter is passed to the function rather than a copy of the value of the parameter.
- The “reference” is the address of the parameter being passed to the function. Rather than a copy of the value being passed to the function, the address of the original parameter is passed to the function.
- Thus, the function is not working with its own copy of the parameter. But rather is sharing the same memory location with the calling function.
- To understand how pass-by-reference works, we need to look more closely at the concept of a pointer and the indirection operator.



# Pointers and the Indirection Operator

- Although we have not made a great deal of fuss about it, you've been using pointers every time you've dealt with a file in C.
- We done something like:

```
FILE *inFilePtr;
```

each time we've set up a file pointer for reading in values from a file. **The \* is the indirection operator in C.**

- Declaring a pointer to an `int`, `float`, or `char` type is done in exactly the same fashion:

```
//declares a pointer to an int
```

```
int *aPtr;
```

```
//declares a pointer to a double
```

```
double *anotherPtr;
```



# Pointers and the Indirection Operator

An aside on the indirection operator `*`:

It might help you to keep things straight when dealing with pointers to understand why the `*` is called an indirection operator. When you refer to a normal variable, such as `int a;`, you are directly referring to the location in memory that has been assigned to hold the values that the variable will represent during the course of the execution of the program. When you refer to a pointer variable, such as `int *ptr;`, you are indirectly referring to the location in memory that has been assigned to hold an integer value. In other words, you are not referring to the location the location itself but to the address of the location that holds the value, thus the indirect reference. This naming convention is a carry over from machine language programming where the direct value of `ptr` is a memory location (address) and the indirect value of `*ptr` is the value at the memory location (address) stored in `ptr`.



# Pointers and the Indirection Operator

Some code

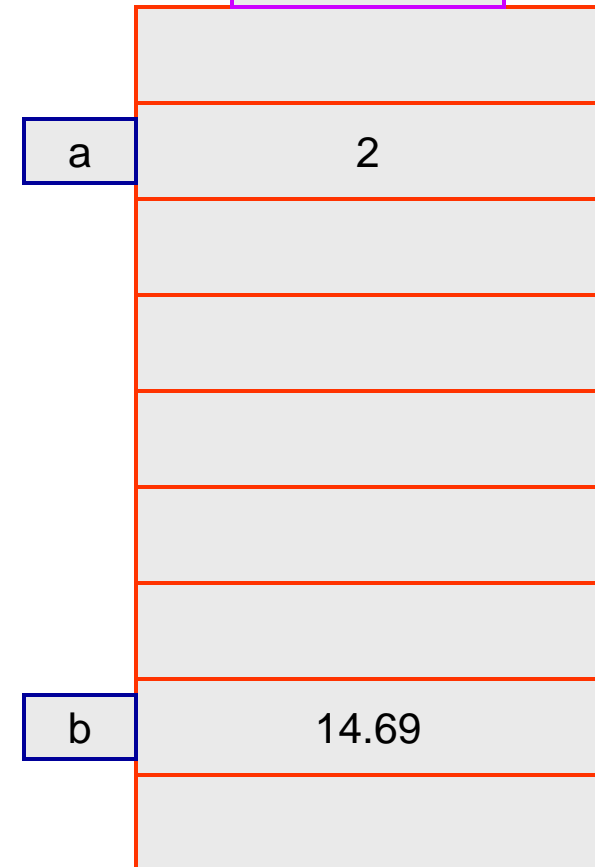
```
int a = 2;  
double b = 14.69;  
int *ptr1;  
double *ptr2;
```

Unlike a normal variable, which if uninitialized assumes the value of the memory location to which it is assigned, a pointer variable literally has no value (i.e., NULL) until it references some specific location in memory.

ptr1 → ?

ptr2 → ?

Memory



# Pointers and the Indirection Operator

- Another way of thinking about the differences between a regular variable and a pointer variable is this:
- When you declare a regular variable, such as `int a;` this means that `a` is capable of having values that are of the `int` type.
- When you declare a pointer variable, such as `int *ptr;` this means that `ptr` is capable of having values that are addresses in the computer memory that hold values that are of the `int` type.
- Thus, pointer variables cannot store any value other than a memory address.



# Pointers and the Indirection Operator

- If we made the following declarations:

```
int a = 18;
```

```
int *ptr;
```

```
ptr = a; //illegal assignment
```

- The above assignment statement would be illegal, since 7 is not a valid memory address that can be accessed by your C program (it is technically an address in the memory but low level addresses are reserved for your OS and are off limits to normal application programs!)



# Pointers and the Indirection Operator

- If we made the following declarations:

```
int a = 18;
```

```
int *ptr;
```

```
ptr = &a; //legal assignment
```

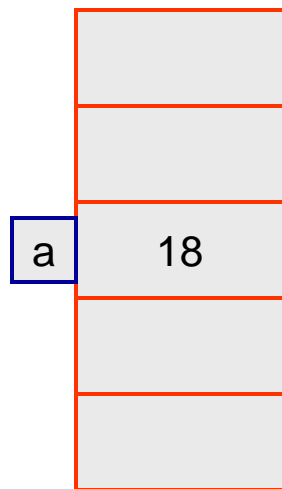
- The above assignment statement assigns the address of the variable `a` (i.e., the address of the memory location that was assigned to the variable `a`) and causes the value of `ptr` to be assigned to that address. (See next page for graphical description of this chunk of code.)



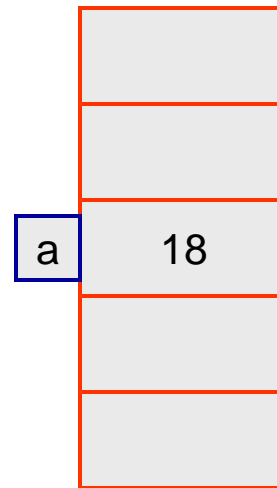
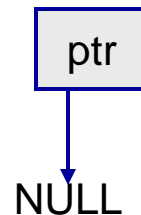


# Pointers and the Indirection Operator

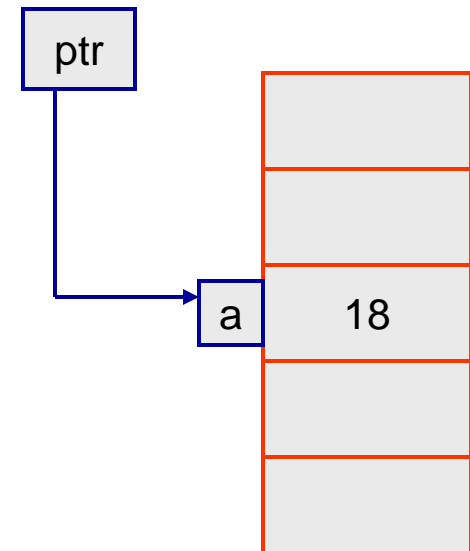
```
1 int a = 18; //declare int variable a set to 18
2 int *ptr; //declare a pointer variable ptr
3 ptr = &a; //assign ptr the address of a
```



Effect of Line 1



Effect of Line 2



Effect of Line 3



# Pointers and the Indirection Operator

- A common use of pointers is to provide access to the value of a variable without reference to the variable itself.
- Think about parameter passing and the difference between sending a copy of a value and sending the address of the value. The function operating with the address of a variable does not need to know the name of that variable, it only needs to be able to access the same memory location.
- This is accomplished using the **indirection operator** (also known as the **dereference operator**).
- The indirection operator can be used to provide access to the memory location referenced by a pointer variable.



# Pointers and the Indirection Operator

- Let's modify the example we were looking at on pages 15-17, to add another couple of lines of code, so that it now will look like the following:

```
1  int a = 18; //declare int variable a set to 18
2  int *ptr; //declare a pointer variable ptr

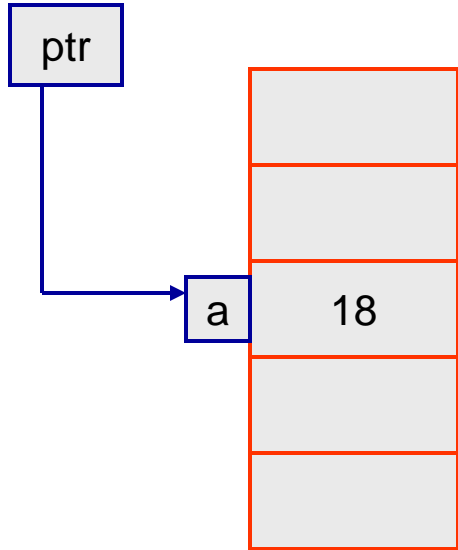
3  ptr = &a; //assign ptr the address of a
4  *ptr = 12; //assigns 12 to location referenced by ptr
5  printf("The value of a is: %d\n", a);
```

- The next page steps through the execution of this code, beginning with line 3.

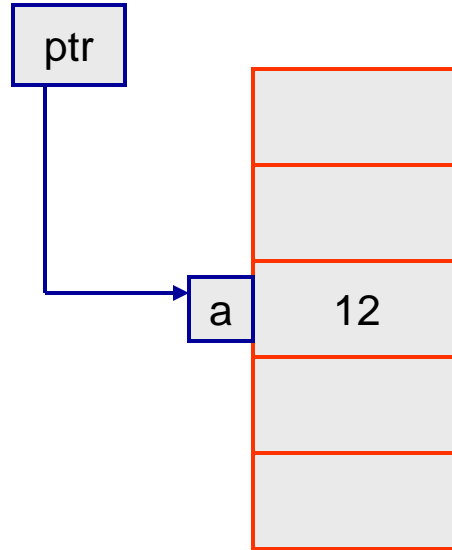


# Pointers and the Indirection Operator

```
3 ptr = &a; //assign ptr the address of a
4 *ptr = 12; //assigns 12 to location referenced by ptr
5 printf("The value of a is: %d\n", a);
```



Effect of Line 3



Effect of Line 4

The value of a is:  
12

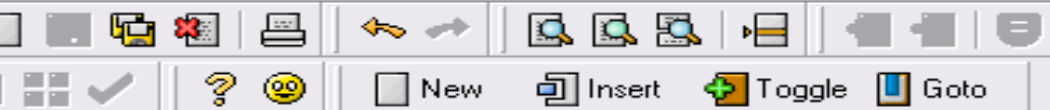
Effect of Line 5



# An Example Done Both Ways

- Let's look at a simple example program that uses a function to compute  $x^3$  for a value passed to the function.
- We'll write the program first using a function where the variable is passed to the function using pass-by-value.
- Then we'll write a second version of the program using a function where the variable is passed to the function using pass-by-reference.
- Note the similarities and differences in the two programs.





cube a number - pass by value version.c | cube a number - pass by reference version.c

```
1 //Functions In C - Part 3 - Program to compute x^3 using pass-by-value
2 //March 1, 2009   Written by: Mark Llewellyn
3
4 #include <stdio.h>
5
6 //function cubeByValue - computes n^3 using pass-by-value
7 int cubeByValue( int n )
8 {
9     return n * n * n;
10 } //end cubeByValue function
11
12 int main()
13 {
14     int number; //user entered input
15     int cubedNumber; //value returned by function
16
17     printf("\nPlease enter an integer value:\n");
18     scanf("%d", &number);
19     //call cubeByValue using pass-by-value parameter
20     cubedNumber = cubeByValue(number);
21     printf( "\nThe value of %d cubed is: %d\n\n", number, cubedNumber);
22     printf("The value of the variable number is still: %d\n\n\n", number);
23
24     system("PAUSE");
25     return 0;
26 } //end main function
27
```

**PASS-BY-VALUE VERSION**

```
K:\COP 3223 - Spring 2009\COP 3223 Program File...  
Please enter an integer value:  
5  
The value of 5 cubed is: 125  
The value of the variable number is still: 5  
Press any key to continue . . .
```



```
1 //Functions In C - Part 3 - Program to compute x^3 using pass-by-reference
2 //March 1, 2009    Written by: Mark Llewellyn
3
4 #include <stdio.h>
5
6 //function cubeByReference - computes x^3 using pass-by-reference
7 void cubeByReference( int *nPtr )
8 {
9     *nPtr = *nPtr * *nPtr * *nPtr; //cubes the value referenced by nPtr
10 }//end cubeByReference function
11
12 int main()
13 {
14     int number; //user entered input
15     printf("\nPlease enter an integer value:\n");
16     scanf("%d", &number);
17     printf("\nThe value of %d cubed is: ", number);
18     // pass address of number to cubeByReference
19     cubeByReference(&number);
20     printf("%d\n\n", number);
21     printf("The value of number is now: %d\n\n\n", number);
22
23     system("PAUSE");
24     return 0;
25 }//end main function
26
```

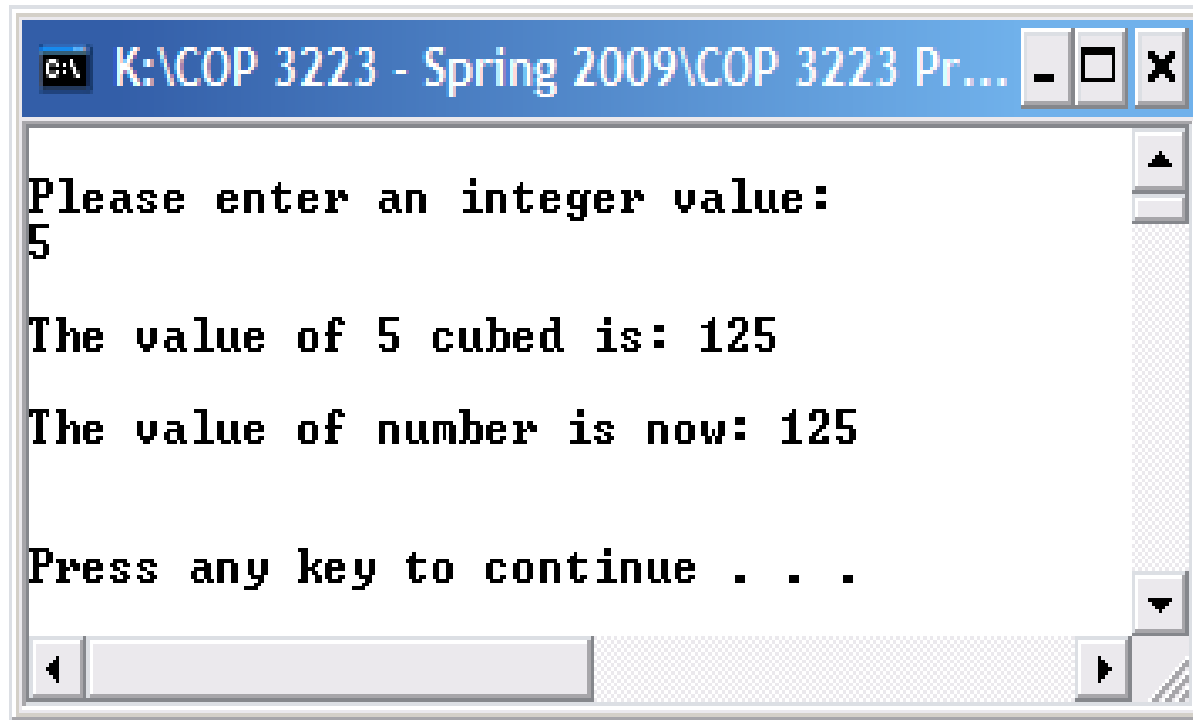
Notice that the function parameter is a pointer variable

## PASS-BY-REFERENCE VERSION

Notice that the address of `number` is passed to the function. The function will modify the value at the address of `number`.







```
K:\COP 3223 - Spring 2009\COP 3223 Pr...  
Please enter an integer value:  
5  
The value of 5 cubed is: 125  
The value of number is now: 125  
Press any key to continue . . .
```



# Practice Problems

1. Trace the execution of the program shown on the next page. Use a technique similar to the one we used to trace the program on page 3.

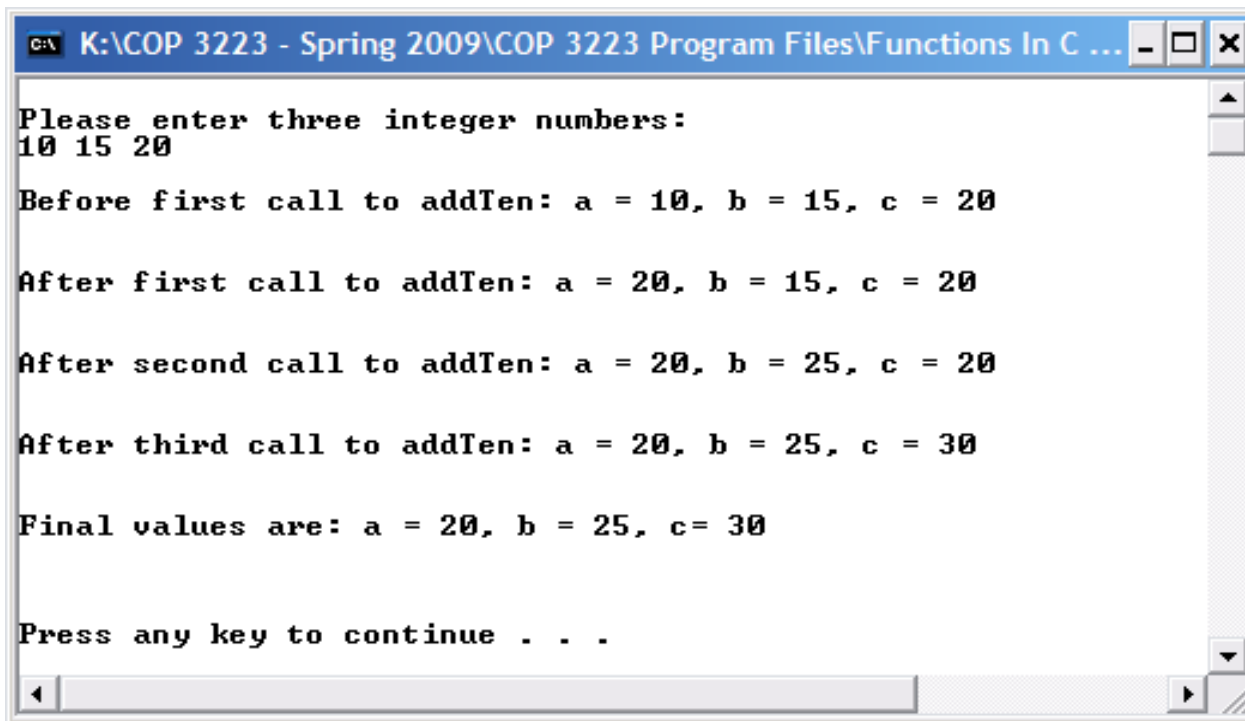


```
5
6 #include <stdio.h>
7
8 int doesSomething(int x, int y, int z)
9 {
10     int sum;
11     sum = z + y + x;
12     if (sum > x * y)
13         return x * y;
14     if (sum == x * y)
15         return y * z;
16     return x * z;
17 } //end doesSomething function
18
19 int main()
20 {
21     int a = 4, b = 1, c = 1;
22     a = doesSomething(a + b, a + c, b + c);
23     printf("\nAfter first call to doesSomething: a = %d b = %d c = %d\n\n", a, b, c);
24     b = doesSomething(a, b, c);
25     printf("After second call to doesSomething: a = %d b = %d c = %d\n\n", a, b, c);
26     c = doesSomething(doesSomething(c, b, a), a, b);
27     printf("After third call to doesSomething: a = %d b = %d c = %d\n", a, b, c);
28     printf("\n\n");
29     system("PAUSE");
30     return 0;
31 } //end main fuction
```



# Practice Problems

2. We mentioned on page 9 that a function in C cannot modify the values of parameters passed to it directly. Write a C program that uses a function with a pass-by-value parameter that will allow you to add 10 to the value of three actual parameters through three separate calls to the function.

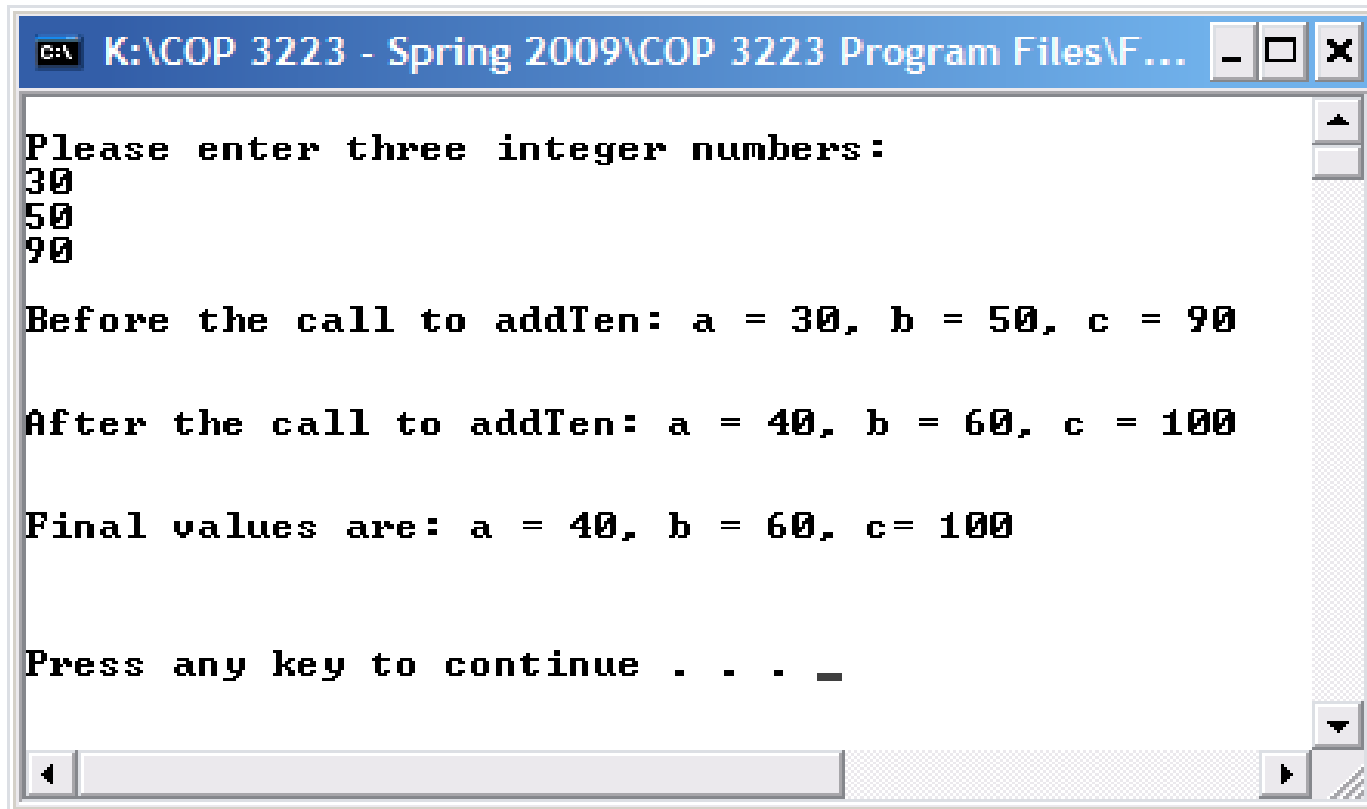


```
K:\COP 3223 - Spring 2009\COP 3223 Program Files\Functions In C ...  
Please enter three integer numbers:  
10 15 20  
Before first call to addTen: a = 10, b = 15, c = 20  
After first call to addTen: a = 20, b = 15, c = 20  
After second call to addTen: a = 20, b = 25, c = 20  
After third call to addTen: a = 20, b = 25, c = 30  
Final values are: a = 20, b = 25, c = 30  
Press any key to continue . . .
```



# Practice Problems

- Rewrite your solution to Practice Problem 2, so that the function now uses the pass-by-reference technique and accepts all three parameters in one call.



```
K:\COP 3223 - Spring 2009\COP 3223 Program Files\F...  
Please enter three integer numbers:  
30  
50  
90  
  
Before the call to addTen: a = 30, b = 50, c = 90  
  
After the call to addTen: a = 40, b = 60, c = 100  
  
Final values are: a = 40, b = 60, c = 100  
  
Press any key to continue . . . _
```

