

# Computer Science & Engineering 150A Problem Solving Using Computers

Lecture 06 - Modular Programming

Christopher M. Bourke

Spring 2009

[cbourke@cse.unl.edu](mailto:cbourke@cse.unl.edu)

## Chapter 6 - Outline

- 6.1 Functions with Simple Output Parameters
- 6.2 Multiple Calls to a Function with I/O Parameters
- 6.3 Scope of Names
- 6.4 Formal Output Parameters as Actual Arguments
- 6.6 Debugging and Testing a Program System
- 6.7 Common Programming Errors

## Overview

- ▶ Chapter 3: separate components – functions – of a program, corresponding to individual steps in a problem solution.
  - ▶ Provide input to a function as *parameters*
  - ▶ Returning (at most) a **single** value (output)
- ▶ Now: we learn how to connect functions to create a program system – an arrangement of parts that makes your program pass information from one function to another.

## Functions with Simple Outputs

`function([argument list])`

- ▶ Argument lists provide the communication links between the *main* function and its functions (sub-programs).
- ▶ Arguments enable a function to manipulate different data each time it is called.
- ▶ So far, we have passed inputs into a function and returned only one result value from a function.
- ▶ We can use output parameters to “return” multiple results from a function.

## Functions – Simple Outputs

How a function call works:

- ▶ When a function call executes, the computer allocates memory space on the *system stack* for each parameter.
- ▶ The value of each actual parameter is stored in the memory cell allocated to its corresponding formal parameter.
- ▶ The function body can access this value *locally*.

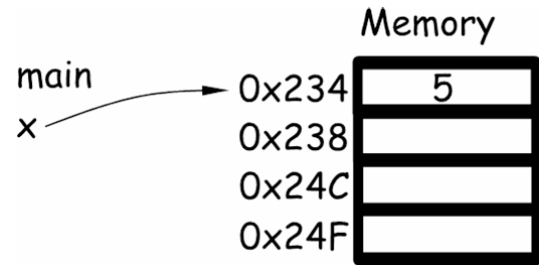
## Example Program

```
1 #include<stdio.h>
2
3 void myFunc(int arg);
4
5 int main(void) {
6     int x = 5;
7     printf("Before calling myFunc, x = %d\n", x);
8     myFunc(x);
9     printf("After calling myFunc, x = %d\n", x);
10    return 0;
11 }
12
13 void myFunc(int arg) {
14     arg = 4;
15     printf("Inside function,      x = %d\n", arg);
16 }
```

## Passing By Value

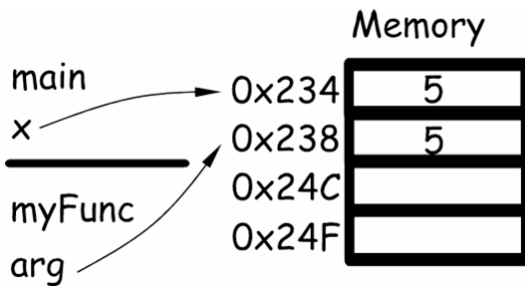
- ▶ This is known as *passing by value*
- ▶ The value of a variable is *copied* into a new memory cell for use by the function that is called
- ▶ Any changes to this new memory cell are *not* reflected in the calling function

## Functions – Simple Outputs



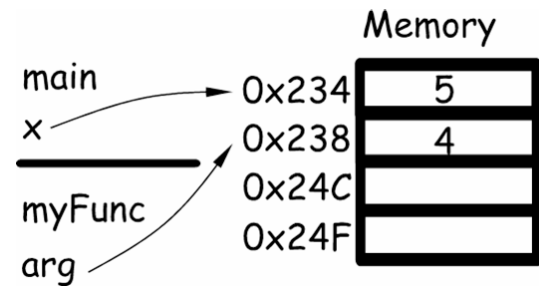
```
1 int main(void) {  
2     int x = 5;  
3     ...  
}
```

## Functions – Simple Outputs



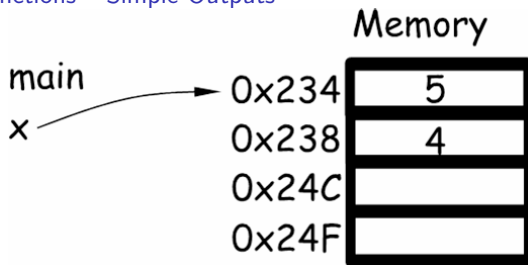
```
1 int x = 5;  
2 myFunc(x);  
3 ...
```

## Functions – Simple Outputs



```
1 void myFunc(int arg) {  
2     arg = 4;  
3 }
```

## Functions – Simple Outputs



```
1 int x = 5;  
2 myFunc(x);  
3 printf("%d\n", x);  
4 }
```

Output = 5

## Pointers

- ▶ How can we allow access to variables when calling functions?
- ▶ The function needs to be told the *memory address* of the variable, not just its value
- ▶ Ultimately, the details are left to the compiler and system, but we do have to have some control in C.
- ▶ A *pointer* is a variable that holds ("points" to) a *memory address*
- ▶ A pointer must point to a memory address that holds a specific *type* of variable (`int`, `double` etc.)

## Pointers I

### Declaring, Assigning

- ▶ Declare a pointer by giving it a type and using the asterisk:  
`int *myIntegerPointer;`  
`double *myDoublePointer;`
- ▶ Initially, you should assign a pointer to point to the NULL memory address: `int *myIntegerPointer = NULL;`
- ▶ Assigning values to a pointer is **not** the same as assigning values to a regular variable:  
`myIntegerPointer = 10;`  
means, that `myIntegerPointer` will now point to the *memory address* 10!

## Pointers II

### Declaring, Assigning

- ▶ The memory address may not belong to the program, accessing or altering whatever is stored at memory address 10 is dangerous!
- ▶ Segmentation Faults, Bus errors, core dumps, etc.

## Pointers

### Dereferencing

- ▶ To store a *value* at a memory address pointed to by a pointer, use the following syntax:  
`*myIntegerPointer = 10;`
- ▶ Now 10 is stored at the memory location pointed to by `myIntegerPointer`
- ▶ This is known as *dereferencing*

## Pointers

### Referencing

- ▶ It is possible to get the memory address of regular variables; this is known as *referencing*
- ▶ Use the ampersand operator to get the memory address of a variable.  

```
1  int k = 2;  
2  int *my_pointer;  
3  my_pointer = &k;
```
- ▶ Now `my_pointer` points to whatever memory address the variable `k` is located at!
- ▶ Changing the value of `k` can be accomplished by using either the variable `k` or by the reference `my_pointer`
- ▶ We have seen this when using `scanf`: the function needs the memory address in order to store the value!

## Pointers

### Referencing - Example

```
1 #include<stdio.h>  
2  
3 int main(void)  
4 {  
5     int k = 10;  
6     printf("The value of k = %d\n", k);  
7     int *aPointerToK = &k; /* this is a pointer to an integer  
8                           type, set to the memory address  
9                           of k */  
10    printf("The memory address of k = %d\n", aPointerToK);  
11    printf("The value stored at that memory address is %d\n", *aPointerToK);  
12    *aPointerToK = 20; /* changing the value of k using its pointer */  
13    printf("The value of k = %d\n",k);  
14 }
```

## Pointers

### Another Example

- ▶ We can use pointers as parameters to functions.
- ▶ Passing the memory address of a variable rather than its value allows the function to access and *change* the variable's value since the function now "knows" where the variable is stored
- ▶ Doesn't just have a copied value
- ▶ This is known as *passing by reference*
- ▶ Demonstration: two functions that swap values

## Pointers I

### Function Arguments

```
1 #include<stdlib.h>
2 #include<stdio.h>
3
4 void swap(int a, int b);
5 void swap_ptr(int *a, int *b);
6
7 int main(int argc, char *argv[])
8 {
9     int a = 42;
10    int b = 2008;
11    printf("a = %d, b = %d\n",a,b);
12    swap(a,b); /* doesn't work! */
13    printf("a = %d, b = %d\n",a,b);
14    swap_ptr(&a,&b);
15    printf("a = %d, b = %d\n",a,b);
16    return 0;
17 }
18 // (Ignore this)
19
```

## Pointers II

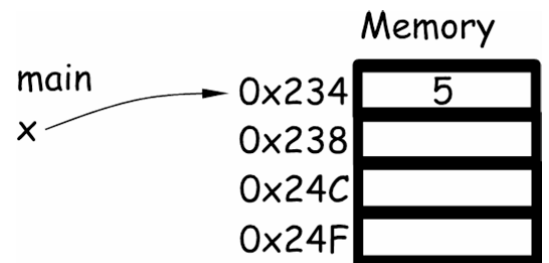
### Function Arguments

```
20
21 void swap(int a, int b)
22 {
23     int temp = a;
24     a = b;
25     b = temp;
26 }
27
28 void swap_ptr(int *a, int *b)
29 {
30     int temp = *a;
31     *a = *b;
32     *b = temp;
33 }
```

## Example Program

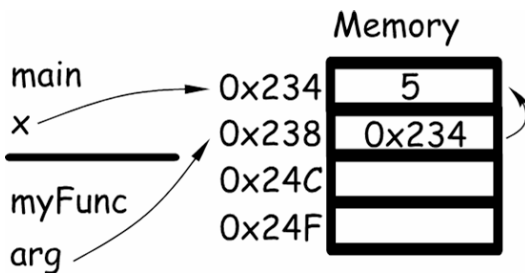
```
1 void myFunc(int *arg);
2
3 int main(void) {
4     int x = 5;
5     myFunc(&x);
6     printf("%d\n", x);
7     return 0;
8 }
9
10 void myFunc(int *arg) {
11     *arg = 4;
12 }
```

## Functions – Simple Outputs



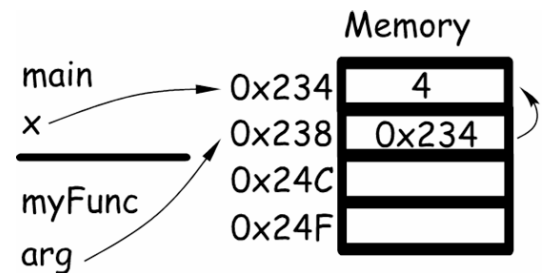
```
1 int main(void) {
2     int x = 5;
3     ...
}
```

## Functions – Simple Outputs



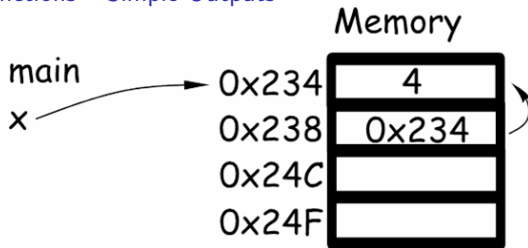
```
1 int x = 5;
2 myFunc(&x);
3 ...
```

## Functions – Simple Outputs



```
1 void myFunc(int *arg) {
2     *arg = 4;
3 }
```

## Functions – Simple Outputs



```
1 int x = 5;
2 myFunc(&x);
3 printf("%d\n", x);
4 }
```

Output = 4

## Returning Multiple Values

- ▶ C only allows *one* value to be returned by any function
- ▶ Sometimes its useful to have a function "return" more than one value
- ▶ Not directly possible, but we can use pointers as function parameters
- ▶ This allows the function to modify the values stored in the passed variables
- ▶ Calling function can then access these values
- ▶ In effect, multiple values are "returned" by the function

## Additional Example I

Sorting Three Numbers

```
1 #include <stdio.h>
2 void order(double *smp, double *lgp);
3
4 int main(void)
5 {
6     double num1, num2, num3;
7
8     printf("Enter three numbers separated by blanks> ");
9     scanf("%lf %lf %lf", &num1, &num2, &num3);
10
11     order(&num1, &num2);
12     order(&num1, &num3);
13     order(&num2, &num3);
14
15     printf("The numbers in ascending order are: ");
16     printf("%.2f %.2f %.2f\n", num1, num2, num3);
17     return 0;
18 }
```

## Additional Example II

Sorting Three Numbers

```
19
20 void order( double *a, double *b) {
21     double temp;
22     if (*a > *b) {
23         temp = *a;
24         *a = *b;
25         *b = temp;
26     }
27 }
```

## Exercise

### Exercise

Recall a previous exercise where we used two functions to compute the roots of a quadratic equation,

$$ax^2 + bx + c$$

using the quadratic equation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Repeat this exercise, but pass  $r_1, r_2$  parameters by reference so that only one function is necessary.

## Answer I

```
1 #include <stdio.h>
2 #include <math.h>
3
4 void computeQuadraticRoots(double a, double b, double c, double *root1, double *root2);
5
6 int main(void)
7 {
8     double a, b, c;
9     printf("Enter a>");
10    scanf("%lf", &a);
11    printf("Enter b>");
12    scanf("%lf", &b);
13    printf("Enter c>");
14    scanf("%lf", &c);
15
16    double r1, r2;
17    computeQuadraticRoots(a, b, c, &r1, &r2);
18    printf("Equation: %.2fx^2 + %.2fx + %.2f\n", a, b, c);
19    printf("1st Root: %f\n", r1);
20    printf("2nd Root: %f\n", r2);
21
22    return 0;
23 }
24
25 /*
26  * We pass root1, root2 by reference so that we can store both values and the calling
27  * function has access to them. Variables a, b, and c are passed by value because the
28  * function does not need to modify them.
29  */
30
```

## Answer II

```
31 void computeQuadraticRoots(double a, double b, double c, double *root1, double *root2)
32 {
33     *root1 = (-b + sqrt(b*b - 4*a*c)) / (2*a);
34     *root2 = (-b - sqrt(b*b - 4*a*c)) / (2*a);
35     return;
36 }
```

## Scope of Names

The **scope** of an identifier (variable name) refers to the region of a program (code block) where the identifier can be referenced (its value accessed).

- ▶ A variable declared inside a function is *local* to that function.
- ▶ Its value cannot be accessed or changed by other functions without a reference to its location
- ▶ You can use the same identifier (variable name) for different local variables in different functions

## Global Variables I

- ▶ Variables declared outside any function are *global* in scope.
- ▶ Any function can access or change their value
- ▶ The same identifier *cannot* be used in any other function
- ▶ Not the same thing as preprocessor macros (`#define PI 3.14`)

## Scope Example

```
1 #include <stdio.h>
2
3 /* global variables */
4 double pi = 3.14;
5 int globalInt = 42;
6
7 void changeVariables();
8
9 int main(void)
10 {
11     int a = 50; /* a is local to main */
12     printf("pi = %f\n", pi);
13     printf("globalInt = %d\n", globalInt);
14     changeVariables();
15     printf("pi = %f\n", pi);
16     printf("globalInt = %d\n", globalInt);
17     return 0;
18 }
19
20 void changeVariables()
21 {
22     int b = 20; /* b is local to changeVariables */
23     /* we cannot access the variable a in this function */
24     /* we can change pi and globalInt here though */
25     pi = 3.1415;
26     globalInt = 21;
27     return;
28 }
```

## Local Scope

- ▶ Variables declared inside nested *code blocks* are local to that code block
- ▶ Cannot be accessed outside the code block even within the same function
- ▶ Relevant when writing loops or conditions: you can declare a variable local to that code block

## Local Scope

```
1 int main(void)
2 {
3     int x = 10; /* local to main, but any
4                 sub-program block can use it */
5     {
6         /* nested program block */
7         printf("inner block: x = %d\n", x);
8         /* y is local to the inner block */
9         int y = 10;
10        printf("inner block: y = %d\n", y);
11    }
12    printf("outer block: x = %d\n", x);
13    printf("outer block: y = %d\n", y); ← illegal
14
15    return 0;
16 }
```

## Using a function to read input

- ▶ Reading input and validating that it is correct can take a non-trivial amount of code.
- ▶ Convenient to use a function to read-validate-reread input
- ▶ Requires pointers

## Example

```
1 #include <stdio.h>
2
3 void getInput(int *evenNumber);
4
5 int main(void)
6 {
7     int myNumber;
8     getInput(&myNumber);
9     printf("Your number: %d\n", myNumber);
10    return 0;
11 }
12
13 void getInput(int *evenNumber)
14 {
15     int error;
16     do {
17         error = 0;
18         printf("Enter an even integer> ");
19         scanf("%d", evenNumber);
20
21         if (*evenNumber % 2 != 0)
22         {
23             printf("You did not enter an even number!\n");
24             error = 1;
25         }
26     } while (error);
27 }
28
```

## Program Notes

- ▶ Notice line 19:  
`scanf("%d", evenNumber);`
- ▶ No ampersand! That's because `evenNumber` is a *pointer*!
- ▶ `scanf` expects a pointer, using the ampersand gives the reference (memory address) of the variable
- ▶ Notice line 21:  
`if(*evenNumber % 2 != 0)`
- ▶ Asterisk dereferences the number, invalid without it!

## Debugging and Testing I

- ▶ As the number of statements in a program grows, the possibility of error also increases.
  - ▶ Reducing the number of operations each function performs also reduces the likelihood of errors.
  - ▶ Small functions are also much easier to read and test.
- ▶ **top-down testing** is the process of testing a program starting at `main` function and testing each function thereafter.
- ▶ We test a function with a **unit test** by writing a short driver function to call it.
  - ▶ This can be done in the main function by commenting out other function calls (comment out the head and legs of stickman to see if the body is correctly made).

## Debugging and Testing II

- ▶ **Bottom-up testing** is the process of separately testing individual functions before inserting them in a program system.
- ▶ **System integration tests** are tests of the entire system
- ▶ Try both top-down and bottom-up to make sure the program is fully tested

## Common Programming Errors I

- ▶ Many opportunities for errors arise when you use functions with parameter lists
  - ▶ Ensuring that the actual argument list has the same number of items as the formal parameter list.
  - ▶ Each input argument must be of a type that can be assigned to its corresponding formal parameter.
  - ▶ An actual output argument must be of the same pointer data type as the corresponding formal parameter.
- ▶ Proper use of parameters is difficult for beginning programmers to master, but it is an essential skill.

## Common Programming Errors II

- ▶ It is easy to introduce errors in a function that produces multiple results.
  - ▶ The output parameter must be of a pointer type (\*)
  - ▶ The calling function neglects to send a correct variable address (using &)
- ▶ An identifier referenced outside of its scope will return an undeclared symbol syntax error.
- ▶ Commonly occurs if brackets {} are misplaced, or if too many open or close brackets are present in the program