

CS 11 Data Structures and Algorithms

Lesson 7: Functions 2

[Skip to Main Content](#)

Section 7.1: Pass-By-Value Versus Pass-By-Reference

Many students struggle to understand the difference between the pass-by-value and pass-by-reference parameter passing mechanisms and when to use each. In section 1 of this lesson I will list 7 strategies for describing the difference between the two. My hope is that those of you who seem to be getting the hang of this will gain an even deeper understanding of the issue by hearing it said in many different ways, and those of you who are struggling will find at least one strategy for describing the difference that makes sense to you.

A quick preliminary note: The choice between pass-by-value and pass-by-reference is made for each individual parameter, not just once for each function. So it is possible to have a function with one or more pass-by-value parameters in addition to one or more pass-by-reference parameters.

Strategy #1) The first strategy for describing the difference is the most practical. You can actually apply this without really understanding pass-by-value and pass-by-reference. The first half of the rule is very easy: **if the value of the parameter doesn't change inside the function, use pass-by-value**. There are three ways to change a value inside a function: an assignment statement with the parameter on the left side of the assignment operator, an extraction operator with the parameter on the right side, or a function call with the parameter as an argument. If none of these three situations appears in the function, then you know that you should use pass-by-value. The second half of this first rule is a little more complex: **if the value of the parameter changes, AND you want the calling function to know about the change, use pass-by-reference**.

Strategy #2) Determine the direction that information is flowing. If the information is flowing from the calling function to the called function, use pass-by-value. If the information is flowing from the called function to the calling function, use pass-by-reference. If the information is flowing in both directions, use pass-by-reference. This is essentially the strategy that I used in explaining the difference originally (in the last lesson).

Strategy #3) Look at the situation from the perspective of the called function. If the information is flowing **in** to the function, use pass-by-value. If the information is flowing **out** of the function, use pass-by-reference. If the information is flowing in both directions, use pass-by-reference. There are a couple of pitfalls with this strategy that I should warn you about. What if the called function sends the parameter down to a third function? Does that make the parameter an "out" parameter? The answer is no. When determining whether a parameter is an "in" parameter or an "out" parameter, you should only consider the interaction between the function in question and the function that called it. This possible confusion is one reason why I choose to describe the difference using Strategy #2 rather than this strategy.

Strategy #4) If the function needs to pass information to the function that called it, use pass-by-reference. Otherwise, use pass-by-value. Okay, now we are starting to say the same thing with just slightly different words. But maybe this will click for someone out there.

Strategy #5) If you use pass-by-reference, the changes made to the parameter inside this function are reflected in the calling function. If you use pass-by-value, they are not.

Strategy #6) In a structure diagram, if the arrows are going down use pass-by-value. If the arrows are going up, use pass-by-reference. A structure diagram is a picture you draw of your program. Each function, including main, is represented with a box. You put main at the top, and each function that main calls is placed below main and connected to main with a line. Then you repeat this process for each of the other functions. Finally, you draw arrows on the diagram for each parameter that is passed. For example, in our draw box program, there would be an arrow starting at getDimensions and going up to main representing the fact that getDimensions is passing the value "width" up to main.

Strategy #7) When you use pass-by-value, the called function receives a value. When you use pass-by-reference, the called function receives a reference, which is a memory address. This is the real behind the

scenes technical explanation of what happens when you pass parameters. More details on this in the next section of this lesson.

Section 7.2: The Real Scoop on Passing Parameters

Note: I apologize for the low tech graphics in this section. I'll fix them up when I have time. Better yet, you could volunteer to create some for me...

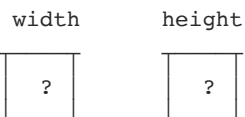
In this section we will be discussing what really goes on behind the scenes when you pass a parameter either by value or by reference. Some might argue that this level of understanding is not necessary when you are just learning a programming language for the first time. I have found, however, that most of my students find that their ability to use parameters correctly is enhanced by understanding the mechanisms involved at a deeper level.

Let's start by tracing through the execution of our draw box program in detail. I'll be drawing lots of repetitive pictures during this process, so the lesson may seem longer than it actually is. You'll probably want to be looking at a copy of the final version of our draw box program while you read through this. At each stage of our trace, I will show you a snapshot of the current situation along with the relevant code.

First we execute the two declaration statements inside main. A declaration statement like `int width;` tells C++ to allocate a memory location somewhere in the computer's memory for a variable called `width`. When drawing this situation, we use a box to represent each memory location. We will use an arrow to keep track of which statements in the program have been executed so far. The arrow will point to the next statement to be executed. A "?" in a box is used to indicate that the value of the variable is unknown, or junk.

```
int main()
{
    int width;
    int height;

    -->getDimensions(width, height);
    drawBox(width, height);
}
```

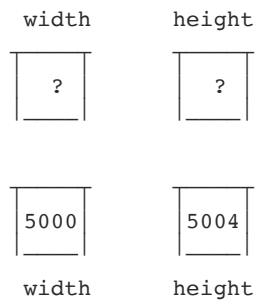


The next thing that happens is the call to `getDimensions`. As we enter `getDimensions`, two parameters are declared, and memory is allocated for these two new variables. If these parameters were pass-by-value parameters, they would get initialized to the values of the corresponding arguments. In this case, the values of the corresponding arguments are unknown, and so the parameters would both get initialized to junk. However, these parameters are not pass-by-value, they are pass-by-reference. The way pass-by-reference works is that instead of being initialized to the value stored in the corresponding argument, the parameters are initialized to the memory address of the corresponding argument. If we assume for a moment that `width` is stored at memory location 5000 and `height` is stored at memory location 5004, the situation would look like this:

```
int main()
{
    int width;
    int height;

    getDimensions(width, height);
    -->drawBox(width, height);
}

void getDimensions(int &width,
                  int &height)
{
    -->cout << "Enter width: ";
    cin >> width;
    cout << "Enter height: ";
    cin >> height;
}
```

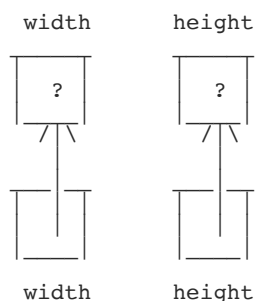


Computer programmers, however, seldom think in terms of the actual value of memory addresses. A memory address is essentially just a way to refer back to that memory location. Instead of using the actual memory address, we will use an arrow to point from the variable where the memory address is being stored to the memory location itself.

```
int main()
{
    int width;
    int height;

    getDimensions(width, height);
    -->drawBox(width, height);
}

void getDimensions(int &width,
                  int &height)
{
    -->cout << "Enter width: ";
    cin >> width;
    cout << "Enter height: ";
    cin >> height;
}
```



Now we have done the hard part: setting up the parameters. We are ready to start executing the statements inside the function `getDimensions`. The first statement doesn't interest us here. The second statement does. Let's say that the user types the value 7 for the width. Where does the 7 go? C++ would normally put it in the local variable `width`, but when we look in `width` we see that it is not a normal variable but rather a "reference" to another variable. So instead of putting the 7 in the local variable `width`, we follow the arrow back to the variable `width` which belongs to `main` and put the value 7 there. It is important to realize that the fact that the names are the same is purely coincidental. C++ does not look at the names, rather it simply follows the arrow and uses the variable that the arrow is pointing to. After following the same procedure for the `cin >> height;` statement, and assuming that the user enters a 4 for the height, our situation is this:

```
int main()
{
    int width;
    int height;

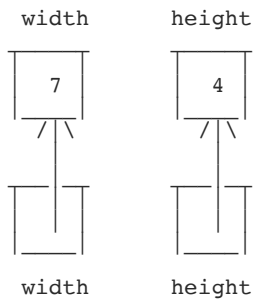
    getDimensions(width, height);
    -->drawBox(width, height);
}

void getDimensions(int &width,
                  int &height)
```

```

{
    cout << "Enter width: ";
    cin >> width;
    cout << "Enter height: ";
    cin >> height;
    -->
}

```



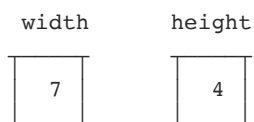
Now we have reached the bottom of the `getDimensions` function. When we reach the bottom of a function, all of the variables declared in that function are deallocated -- for now just think of them as going away. So when we get back to `main` after having executed the `getDimensions` function, our situation looks like this:

```

int main()
{
    int width;
    int height;

    getDimensions(width, height);
    --> drawBox(width, height);
}

```



The rest of the program involves only pass-by-value. There are no more examples of pass-by-reference. We won't trace through all of the rest of the program, but we will do a bit more for the sake of illustration.

The next statement to be executed is the call to `drawBox`. Two values are sent. The first value is the value stored in `width` (7), and the second value is the value stored in `height` (4). As we enter `drawBox`, two parameters are declared: `width` and `height`. Since `width` is listed first, it gets initialized to the first value that was sent (7). Since `height` is listed second, it gets initialized to the second value that was sent (4). Notice that this has nothing to do with the names of the arguments or parameters, but with the order in which they are listed. Here is the picture at this point:

```

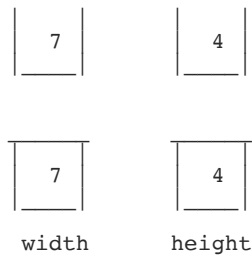
int main()
{
    int width;
    int height;

    getDimensions(width, height);
    drawBox(width, height);
    -->
}

void drawBox(int width, int height)
{
    drawHorizontalLine(width);
    draw2VerticalLines(width - 2,
                       height - 2);
    drawHorizontalLine(width);
}

```





Now we are ready to execute the statements inside `drawBox`. When we call `drawHorizontalLine`, the value that will be sent is 7, since that is the value stored in `width`. When we call `draw2VerticalLines`, the values that will be sent are 5 and 2, the values of `width - 2` and `height - 2`, respectively. And so on.

I have not yet said anything about global variables. Global variables are variables that are declared outside of any function and that can, therefore, be accessed from any function. You should not use global variables, but you should know how they behave in case you have to deal with someone else's code that uses global variables. The rule is this. When a variable name occurs in any function (including `main`), C++ first looks to see if there is a variable with that name that is local (i.e. declared inside that function). If so, that is the variable that is used. If not, C++ proceeds to look for a global variable with that name. If there is a global variable with that name, it is used. If there is neither a local variable or a global variable with that name, a compile-time error results. One result of this procedure is that if there is a global variable AND a local variable with that name, it is the local version that is used, not the global version.

On an exam for this course you will be asked to trace through a very confusing program which involves pass-by-value and pass-by-reference. You will be required to predict what the output of this program will be. In order to help you prepare for this question, I have provided 6 trace programs. You can find links to these programs below. I strongly suggest that you print these out, try to predict the output, and then compile and run them to see if your answers are correct. You will have to deal with global variables on a few of the trace programs I have provided, and you may see global variables on trace problems on exams.

Note about the trace problems: Some of the output from these trace programs will be "junk", since the variables will be uninitialized. Be careful: sometimes this junk will be obviously junk (say you get a result like -80343943). Other times, the junk might be a number like "0", which looks like a real value that was computed, but is really just the value that happened to be in that variable.

trace example 1

trace example 2

trace example 3

trace example 4

trace example 5

trace example 6

Section 7.3: Generating Random Numbers

You will probably need to know how to generate random numbers to do at least one of your assignments in this class. This is done in C++ with the function `rand()`. This function is a value returning function, which means that it is used as an expression when you call it. It returns a random integer between 0 and something very big (roughly 32,000). The following program illustrates the use of the `rand()` function and shows the results when I ran the program. Notice that in order to use the `rand()` function I have to put a `#include <cstdlib>` at the top of my program.

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

int main()
{
    for (int i = 0; i < 100; i++){
        cout << setw(8) << rand();
        if (i % 5 == 4){
            cout << endl;
        }
    }
}
```

}

16838	5758	10113	17515	31051
5627	23010	7419	16212	4086
2749	12767	9084	12060	32225
17543	25089	21183	25137	25566
26966	4978	20495	10311	11367
30054	17031	13145	19882	25736
30524	28505	28394	22102	24851
19067	12754	11653	6561	27096
13628	15188	32085	4143	6967
31406	24165	13403	25562	24834
31353	920	10444	24803	7962
19318	1422	31327	10457	1945
14479	29983	18751	3894	18670
8259	16248	7757	15629	13306
28606	13990	11738	12516	1414
5262	17116	22825	3181	13134
25343	8022	11233	7536	9760
9979	29071	1201	21336	13061
22160	24005	30729	7644	27475
31693	25514	14139	22088	26521

Typically we don't want numbers between 0 and 32 thousand something. Let's suppose we want numbers between 0 and 100. So how can we scale these numbers so that we get a result that is between 0 and 100? Let's use the modulus (%) operator. If I change the line

```
cout << setw(8) << rand();
```

to

```
cout << setw(8) << rand() % 101;
```

what will I get? Well, if I take any number % 101, what is the smallest result I could get? 0. What is the largest number I could get? 100. Exactly what I wanted. The following program illustrates this:

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

int main()
{
    for (int i = 0; i < 100; i++){
        cout << setw(8) << rand() % 101;
        if (i % 5 == 4){
            cout << endl;
        }
    }
}
```

72	1	13	42	44
72	83	46	52	46
22	41	95	41	6
70	41	74	89	13
100	29	93	9	55
57	63	15	86	82
22	23	13	84	5
79	28	38	97	28
94	38	68	2	99
96	26	71	9	89
43	11	41	58	84
27	8	17	54	26
36	87	66	56	86
78	88	81	75	75
23	52	22	93	0
10	47	100	50	4
93	43	22	62	64
81	84	90	25	32
41	68	25	69	3
80	62	100	70	59

Now we have random numbers between 0 and 100.

There are several other points to keep in mind when working with random numbers.

1. Usually it won't be enough to just print out a random number. You'll want to save the random number in a variable and then use that variable instead of directly printing out the random number. This way the variable will still be available to be used later. So instead of just printing the random number, you might have a statement like `num1 = rand() % 101;`.
2. Notice that the number I put after the % is always ONE BIGGER than the high end of the range of numbers I want to generate. So if I want to generate numbers between 0 and x, I would use a statement like `num1 = rand() % (x + 1);`.
3. What I have told you so far will work fine for generating random numbers, but each time you run your program you will get the same sequence of random numbers. Here's a little trick to make it so you get a different sequence of random numbers each time you run your program. First, put the statement `#include <ctime>` at the top of your program. Then put the statement `srand(static_cast<unsigned>(time(0)));` as the first line in your main function. Make sure the argument is a zero, not an upper case letter 'O'. This function call uses the clock time to initialize the random number generator. Simply put it as the first line in your main function and forget about it. Don't put it anywhere else in your program.

© 2010 - 2016 Dave Harden