CS 11 Data Structures and Algorithms

Lesson 6: Functions 1

Skip to Main Content

Section 6.1: Introduction to Functions

Black Boxes

Previously in the lessons we discussed the importance of writing **good** computer programs. Among the goals that we have for our programs as we write them are that we want them to be:

1. easy to read and understand
2. easy to debug
3. easy to modify
4. easy to reuse parts in another program

In this chapter we will be looking at functions. The use of functions is one of the most powerful methods we have for accomplishing the four goals listed above. We use functions to break a program down into smaller parts. You could think of each of these parts as a "module". As an analogy, let's think about electronic equipment. These days when you have your television or stereo repaired, no one actually goes in and messes around with individual wires. The equipment is made up of modules. If a module breaks, the repairperson removes it and replaces it with a new module. He doesn't need to know what is inside the module or how it works. He only needs to know what the module does.

(Actually, I have no idea whether this is really an accurate description of how electronics are repaired. It makes a great analogy, though.)

We would like our programs to work kind of like this. We want to break the program up into "modules," called functions. This way, once we write a function and debug it and test it, we can forget about how it works and what's inside, and just remember what it does. This is often called the "black box" analogy. Once we have a particular function working perfectly, we treat it as a black box; that is, we imagine that we can't look inside to see what the details are of how it works, we just know what it does. We will see later that our black boxes (i.e. functions) can have carefully controlled information going in and out of them, and what a black box (i.e. function) does is often specified according to what goes in and what comes out.

An important thing to notice about these black boxes is that they are completely self-contained and independent. If one black box has access to what's inside another one, we can no longer treat them independently. In order for the black box concept to work, we have to be able to change how a black box works on the inside and know that the rest of our program will still work.

Top-Down Programming

In order to use this new "black box" strategy, we are going to have to change the way we look at programming a bit. Until now when we have gone to write a program we have written down every little detailed step, step by step, until we reached the end of the program. What we want to do now is look at the problem and instead of asking ourselves what the first (detailed) step is, and then what the next (detailed) step is, and then what the next (detailed) step is, and so on, we want to ask ourselves "what are the 3 or 4 or 5 major steps involved in accomplishing this task?" Once we have them down, then we go back and fill in the details. This is called **top-down programming** or **stepwise refinement.**

Suppose we are writing a computer program to bake a cake. By our old methods we would have started our program off something like this:

1) open cupboard door
2) get out baking pan
3) open refrigerator door
4) remove 1 egg
5) get out a small bowl
6) open the cupboard door

7) measure out one cup of flour
.
.
.
.
192) open oven door
193) insert baking pan
194) close oven door
195) wait 45 minutes
.
.
.

Using stepwise refinement, we would approach the problem like this:

1) gather ingredients
2) mix ingredients
3) bake

Then we would have to go back and define more precisely what we mean by each of these three steps.

The Advantages of Using Functions:

1) Our programs become easier to read. A long complicated program written without functions can be a gigantic task to try to read and understand. All of the details are right there in front of you and it's difficult to get the big picture of what the program is all about. If we use functions effectively, it becomes trivial to understand each individual function. Instead of being a huge thing with hundreds of confusing statements, the main function can become a list of three or four or five easy to understand steps. If we want more details about one (or more) of the individual steps, we can go look at the function definitions.

2) Our programs become easier to debug. Imagine that we have a huge program all written out in one huge main function. Further assume that all the different parts of the program are constantly messing with numerous different variables. If there is a problem with our program, it will be very difficult to isolate the problem and fix it. Furthermore, even after we find the place in the program where things are going wrong, it is likely that in the process of fixing the problem we will accidentally introduce problems elsewhere in the program, because we have no guarantee that a change made in this part of the program won't affect something going on in another part of the program. What a nightmare! However, if we use functions, it is easy to go through the functions one by one, making sure that they work correctly, until we find the one that is defective. Then we can forget about the rest of our program and focus on that one function. Again, it is important to keep our functions self-contained and independent, so we know that no other part of our program will be messed up because of the change we are making here.

3) Our programs become easier to modify. When we use functions and we want to make a modification to our program it is easy to locate the function that needs to be changed. As we make the modification, we don't have to worry about whether our changes will adversely affect some other part of our program, because our functions are independent.

4) Parts of our programs become easier to reuse. Now that we have split our program up into functions, if we find ourselves writing another program where we need to do a similar task, we can easily remove the function from one program and insert it into the new program. We don't have to concern ourselves with what is inside the function, only with what the function does. The extent to which this is true can be greatly influenced by how we write our functions and what names we choose for our functions. During the discussion that follows, we will be careful to write and name functions to maximize this benefit of using functions.

Section 6.2: Using Functions in C++

Well, it's time we actually saw a concrete example of using functions. Let's write a program that produces the following output:

```
XXXXXX
X    X
X    X
XXXXXX
```

Although this is a very specific picture we are drawing, let's write a solution that is "general"; that is, we would like to make it so that by simply changing one number in our program we could change the width or the height of this picture.

Additional note: for the sake of illustration, we will be writing our program without the setw function, even though that function could simplify our code.

Before studying functions we might have started by thinking about the details: how will we print each individual "X"? Instead, we begin by breaking the problem down into steps. Look at the box for a minute. How would you break the task down into 2 or 3 or 4 steps? I would suggest breaking it down into the sections marked by the horizontal lines in this picture:

```
XXXXXX
X    X
X    X
XXXXXX
```

What would you choose for the names of the functions to perform each of the three subtasks we have illustrated here? A common first suggestion goes something like this:

1) do top
2) do middle
3) do bottom

There are some problems with the names of these functions. First, they aren't very precise. What do we mean by "do"? Let's replace the word "do" with the word "draw".

Furthermore, if we choose these names for our functions we will be forced to write two different functions, one called "drawTop" and one called "drawBottom". But these two functions would do exactly the same thing! The next suggestion is often to simply call "drawTop" twice, like this:

1) drawTop
2) drawMiddle
3) drawTop

This still presents a problem. Now when we look at our main function it is confusing. If the task is to draw a box, why would we want to draw the top two times? The solution to this dilemma is to give our functions names that are not tied to the context of the program as a whole (drawTop only makes sense because the program is drawing a box!) but instead simply describe what the function does. In this case, the function in question draws a horizontal line. So we will call it "drawHorizontalLine".

What about step 2? Can we think of a name for that function that doesn't refer to a part of a box but instead simply describes what the function does? If you look carefully at the picture above, you will notice that the middle section of the box is really just two vertical lines. So we will call our second function "draw2VerticalLines". As you can see, it is important to think carefully about the names of your functions.

In conclusion, our steps will be:

1) drawHorizontalLine
2) draw2VerticalLines
3) drawHorizontalLine

Putting these into an actual program would look like this:

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

int main()
{
    drawHorizontalLine();
    draw2VerticalLines();
    drawHorizontalLine();
}
```

The three lines in this main function are called "function calls". Notice that they look just like statements in that they appear on a line all by themselves and have a semi-colon at the end. Also notice that each function

call is followed by a set of parentheses. This will always be the case in C++. What we mean when we write a function call as a statement is "go execute all of the statements in the function called, and then return to this point in the program when you are done".

We are now done with our program except that we still have to define `drawHorizontalLine` and `draw2VerticalLines`. Here's the definition of `drawHorizontalLine`:

```
void drawHorizontalLine()
{
    int count;
    for (count = 0; count < 6; count++){
        cout << "X";
    }
    cout << endl;
}
```

Notice that if we want to change the width of the box from 6 to some other number, the only change we would have to make in this function would be to change the 6. If we were using good programming style this 6 would be a named constant, but we are going to leave it as a 6 for now. Later on in this lesson that will change.

Section 6.3: Defining draw2VerticalLines()

The definition of `draw2VerticalLines` is a bit more involved. Once again, we need to postpone thinking about the details right away. We start by realizing that the task involves printing out 2 rows. This leads us to a partial solution:

```
void draw2VerticalLines()
{
    int rowCount;

    for (rowCount = 0; rowCount < 2; rowCount++){

        <insert code to print one row>

    }
}
```

Each row will be a pair of X's with some spaces (4 in this case, since the width is 6), like this:

```
X    X
```

Now we have a couple of options. One option would be to write the code to print one row and insert it right into this function. A better choice would be to make a new function called "drawOneRow." In general, anytime you can name a task which a sequence of statements is performing it is a good idea to make the sequence of statements into a function. Notice that if we had chosen to insert the code right here, we would have nested loops. Using a function call here is nice because nested loops tend to be difficult to read.

Here is the code for drawOneRow:

```
void drawOneRow()
{
    int spaceCount;

    cout << "X";
    for (spaceCount = 0; spaceCount < 4; spaceCount++){
        cout << " ";
    }
    cout << "X" << endl;
}
```

Below you will find the code for our complete "draw box" program. There are several things to notice. First, there are three distinct occurrences of each function name. For example

```
draw2VerticalLines()
```

occurs three times:

- It occurs once in the main function. In this case, we place it there as an instruction. We are telling C++ to go execute the function. It is called a **function call**. Notice that the word void does not appear, and the line ends with a semi-colon.

- It occurs once toward the bottom of the program where we define it. Not surprisingly, it is called a **function definition** in this case. The first line of a function definition is called the **function header**. Notice that the word void is required, but there is no semi-colon at the end of the line.

- It occurs once at the very top of the program. In this case it is called a **function prototype** or (less commonly) **function declaration**. Prototypes are used to let C++ know that it's okay if we call a function called `draw2VerticalLines` because we are going to define one later on. If we forget to include the prototype, C++ will stop when it reaches the function call and say "sorry, the function `draw2VerticalLines` is not recognized." Notice that the word void is required AND the line ends with a semi-colon.

Doing the prototypes is very easy. When you are done with your program, but before you try to run it, simply copy each function header in your program and paste them all at the top (right after your include statement). Just make sure that you end each one with a semi-colon.

It is important when writing functions to leave lots of whitespace between function definitions. It makes it much easier for someone reading your program to identify individual functions and to see where one function ends and the next one begins. In programs that you are turning in, you should put at least 6 blank lines between function definitions (and no, comments are not blank lines). When I was getting my Computer Science degree, the rule of thumb was each function should be on a separate sheet of paper (yes, we used to submit actual printouts of our programs...). Until recently, I required 2 inches of whitespace between functions. So requiring only 6 blank lines represents a significant "easing up" on this requirement.

In order to save space, future examples in these notes may not employ this technique; however, you should. Don't lose points for something as easy to get right as this.

You are now ready to have a deeper understanding of what `main` is all about. `main` is simply a function. The only difference between `main` and any other function in your program is that `main` happens to be where C++ begins execution. Although the convention is to place it first among the functions in a program (and this is good style), there is no requirement of this. Functions in C++, including `main`, are all equal, self-contained, and independent.

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

void drawHorizontalLine();    // these 3 lines are
void draw2VerticalLines();    // called "prototypes"
void drawOneRow();

int main()
{
    drawHorizontalLine();    // these three lines are
    draw2VerticalLines();    // "function calls"
    drawHorizontalLine();
}
```

```
void drawHorizontalLine()    // this is a
{                            // "function definition"
    int count;

    for (count = 0; count < 6; count++){
        cout << "X";
    }
    cout << endl;
}
```

```
void draw2VerticalLines()    // this is another
{                            // "function definition"
    int rowCount;

    for (rowCount = 0; rowCount < 2; rowCount++){
        drawOneRow();
    }
}
```

```
void drawOneRow()
{
    int spaceCount;

    cout << "X";
    for (spaceCount = 0; spaceCount < 4; spaceCount++){
        cout << " ";
    }
```

```
        cout << "X" << endl;
    }
```

Section 6.4: Parameters and Arguments

Let's take another look at the program that we wrote in section 6.3. Instead of always drawing a rectangle with a width of 6 and a height of 4, let's modify the program so that the user enters the height and width of the rectangle. Here's what our new main function might look like:

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

        cout << "Enter width: ";
        cin >> width;
        cout << "Enter height: ";
        cin >> height;

        drawHorizontalLine();
        draw2VerticalLines();
        drawHorizontalLine();
    }
```

Taking a look at our drawHorizontalLine function, it looks like we only need to change the `6` to `width`, like this:

```
    void drawHorizontalLine()
    {
        int count;
        for (count = 0; count < width; count++){
            cout << "X";
        }
        cout << endl;
    }
```

This is not a bad first try. The problem is, we have violated our most important rule about functions: that they should be self-contained and independent. The way we have written this, we are assuming that the value of `width` used in `drawHorizontalLine` will get its value by looking back at the value of `width` in the main function. But what if we take `drawHorizontalLine` out of this program and try using it in another program? `drawHorizontalLine` is no longer self-contained: it requires that `width` be carefully set to an appropriate value before `drawHorizontalLine` is called. As another example, what if we were dealing with a large, involved program and some other function somewhere inadvertently changed the value of width before `drawHorizontalLine` got called? The way we have written this program, even if we did get the program to work correctly, it would be difficult to modify because every time we change a variable we have to check with all the other functions in our program to make sure that it is ok that we change the variable, and that we won't interfere with something that some other function is doing.

As it turns out, the strategy we used for the program above won't even work in C++. But we have a problem. *Somehow* `drawHorizontalLine` needs to find out what the width of the box should be. In order to deal with this kind of situation, where several functions need to have access to the same information, C++ has set up a very careful system of communicating between functions.

When `main` calls `drawHorizontalLine`, it needs to tell `drawHorizontalLine` how many "X"s to print out. In order to do this, we put the value that `main` is communicating to `drawHorizontalLine`, `width` in this case, in parentheses after the function call:

```
    drawHorizontalLine(width);
```

The way to interpret the function call now is that when `main` calls `drawHorizontalLine` it is telling it to do its job and at the same time is sending a value -- whatever the value is that happens to be stored in the variable `width`. When we put a variable in parentheses after a function call like this the variable is called an **argument**.

Let's look at the situation from the perspective of `drawHorizontalLine` now. `drawHorizontalLine` needs to inform C++ that when it gets called it must be sent a value. We indicate this by putting a variable declaration in the parentheses after the words `drawHorizontalLine` in the function definition header:

```
    void drawHorizontalLine(int numXs)
```

This variable declaration is called a **parameter**. The mechanism works like this: When `drawHorizontalLine` gets called, the function that called it must send a value. When `drawHorizontalLine` begins execution, the variable that is declared in the parentheses is initialized to this value. For example, if the function that calls `drawHorizontalLine` sends the value 7, then the variable `numXs` gets initialized to 7. The words `int numXs` in this example are really a variable declaration, just the same as if we had declared the variable on the first line of the function itself. The only difference is that local variables normally start with no initial value, but because `numXs` is a parameter, it will begin with an initial value.

Here is the correct version of `main` and `drawHorizontalLine`:

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

    cout << "Enter width: ";
    cin >> width;
    cout << "Enter height: ";
    cin >> height;

    drawHorizontalLine(width);
    draw2VerticalLines();
    drawHorizontalLine(width);
}
```

```
void drawHorizontalLine(int numXs)
{
    int count;

    for (count = 0; count < numXs; count++){
        cout << "X";
    }
    cout << endl;
}
```

Section 6.5: How to View Functions and Parameters

Even though when we look at this whole program at once, we see that the variable width in main and the variable numXs in drawHorizontalLine look like they are the same thing, we have carefully chosen to give them different names. This is to illustrate the fact that **they are not the same variable!** This example leads us to the following two important rules of how to view functions and parameters:

1) When writing a function always look at the situation from the perspective of the function you are writing.

One way of doing this is to pretend that you *are* that function. For example, when we are writing the function drawHorizontalLine, we say to ourselves, "if I am function drawHorizontalLine, what do I need to know to do my job? Well, I need to know how many Xs to print out!" Notice that when we are looking at the situation from the perspective of the function drawHorizontalLine, we don't even realize that we are drawing a box! That is why it wouldn't make any sense to call the parameter width instead of numXs. Our job doesn't have anything to do with boxes, it only has to do with drawing a horizontal line. An advantage of this approach, besides the simple fact that it keeps our function independent of the rest of the program, is that now if we want to use the drawHorizontalLine in another program that doesn't have anything to do with boxes, it still makes sense: in order to do our job, we need to know how many Xs to print out. If you are tempted to call the parameter width instead of numXs, it probably means that you are still thinking about the whole program as one entity instead of thinking of functions as distinct entities that should be self-contained and not dependent on the rest of the program.

Now let's switch and look at the situation from the perspective of main. Main says, "ok, I want drawHorizontalLine to do its job, but it needs me to send it one value -- the value that tells it how many Xs to print out. I'm storing the number of Xs that I want printed out in my variable width, so that is the value I will send." So, in main, width goes in the parentheses because that is the correct value to send.

2) Think of parameter passing as communication, not as a sharing of variables.

If you start thinking of width and numXs as being different names for the same variable you will get in big trouble. They are not the same variable. NumXs is simply the place where drawHorizontalLine stores whatever value got sent.

It is extremely important to realize that what is being passed from one function to another here is **not** a variable. It is simply a value. In fact, it would be perfectly legal to put a value or even an expression in the parentheses. For example we could have a program where we called the function drawHorizontalLine like this:

```
drawHorizontalLine(3);
```

which would mean to print 3 Xs. Or we could say

```
drawHorizontalLine(2*width-1);
```

so that, for example if width was 7, drawHorizontalLine would print out 13 Xs.

Because what is being passed is a value and not a variable, this parameter passing mechanism is called **parameter passing by value**.

Section 6.6: Defining draw2VerticalLines with Parameters

Let's get back to our drawBox program. We haven't finished it yet: we have only fixed drawHorizontalLine. We still have to fix draw2VerticalLines. We begin by following rule number 1 from above and looking at the situation from the perspective of the draw2VerticalLines function. We ask ourselves, "ok, if I'm function draw2VerticalLines, and my whole purpose in life is to draw two parallel vertical lines of starts when I am called, what information do I need in order to do my job? Well, I need to know how many spaces to put between the two lines, and I need to know how many horizontal rows to print." A horizontal row is this:

```
X    X
```

So, for example, 4 rows would look like this:

```
X    X
X    X
X    X
X    X
```

Now we choose names for the two pieces of information that we need. We'll call the number of spaces numSpaces and the number of horizontal rows numRows. These two variables will go in the parentheses in our function header. They will be our parameters. After going through a similar process to decide what the parameters for drawOneRow should look like, the result will be this:

```
void draw2VerticalLines(int numSpaces, int numRows)
{
    int rowCount;

    for (rowCount = 0; rowCount < numRows; rowCount++){
        drawOneRow(numSpaces);
    }
}
```

```
void drawOneRow(int numSpaces)
{
    int spaceCount;

    cout << "X";
    for (spaceCount = 0; spaceCount < numSpaces; spaceCount++){
        cout << " ";
    }
    cout << "X" << endl;
}
```

To complete our task now we have only to make the necessary changes inside main to match the changes we have made in draw2VerticalLines. To do this we must once again switch to looking at the situation from main's perspective. From main's perspective, draw2VerticalLines is asking for two pieces of information: the number of spaces and the number of rows. How many spaces does main want draw2VerticalLines to print? The answer is "2 less than the width of the box," or width — 2. How many rows does main want draw2VerticalLines to print? The answer is similar: "2 less than the height of the box," or height — 2. So these two expressions will be placed in the parentheses after the call to function draw2VerticalLines.

Here (finally) is a complete version of our new program which allows the user to input the height and width of the box to be drawn. Notice that the prototypes must be updated when we add parameters to our function

headers, but the same rules about prototypes still apply: simply copy the actual function header and paste it at the top.

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

void drawHorizontalLine(int numXs);
void draw2VerticalLines(int numSpaces, int numRows);
void drawOneRow(int numSpaces);

int main()
{
    int width;
    int height;

    cout << "Enter width: ";
    cin >> width;
    cout << "Enter height: ";
    cin >> height;

    drawHorizontalLine(width);
    draw2VerticalLines(width - 2, height - 2);
    drawHorizontalLine(width);
}
```

```
void drawHorizontalLine(int numXs)
{
    int count;

    for (count = 0; count < numXs; count++){
        cout << "X";
    }
    cout << endl;
}
```

```
void draw2VerticalLines(int numSpaces, int numRows)
{
    int rowCount;

    for (rowCount = 0; rowCount < numRows; rowCount++){
        drawOneRow(numSpaces);
    }
}
```

```
void drawOneRow(int numSpaces)
{
    int spaceCount;

    cout << "X";
    for (spaceCount = 0; spaceCount < numSpaces; spaceCount++){
        cout << " ";
    }
    cout << "X" << endl;
}
```

Declaring Variables Locally

You are now ready to learn a very important rule. I think this is the most important rule you will learn in this class. Here it is:

> Every variable must be declared inside the function in which it is used.

The word **local** means "inside the function in which it is used." So, a more concise way of saying this rule is

> Every variable must be declared locally.

To illustrate this rule, let's look back up at the function draw2VerticalLines. There are a total of three variables used in that function: rowCount, numRows, and numSpaces. The first one is easy: it is declared just inside the function using our normal declaration statement. What about numRows and numSpaces?? As was mentioned (briefly) before, the parameters are really just special declaration statements. They are special because the variables that are being declared get initialized. But they still count as declarations. So all three of the variables used in draw2VerticalLines have been declared.

A corollary to this rule is that you should never use global variables. C++ does allow you to declare a variable outside of any function, just like you can declare constants outside of any function. When you do this it is called a global variable, and is accessible from any function. The problem is that if we have global variables, our functions are no longer independent and self-contained. Don't use them.

Section 6.7: Pass-By-Reference

What we studied in the last three sections of this lesson was a parameter passing mechanism called pass-by-value. In order to illustrate the use of the second (and last) parameter passing mechanism, we will try to improve our draw box program by restructuring it. Our goal here is not to change the behavior (output) of our program in any way, but to restructure it. We have a main function which has too much detail in it. The first 4 statements in main accomplish the task "getDimensions". The second 3 statements in main accomplish the task "drawBox". So let's change our main function so it looks like this:

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

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

The next step, then, would be to take those seven statements that we removed from main and put them in the appropriate new function that we are creating:

```
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;
}
```

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

There is a problem with this though. drawBox will work fine the way it is. Just like with all of the other functions we have defined, drawBox is receiving values with which to initialize its parameters. However, if you look at getDimensions carefully the situation is different. getDimensions is not receiving values with which to initialize its parameters. On the contrary, getDimensions would like to start out with empty parameters, fill them up with the users input, **and then pass this information back to main!** We cannot do this with pass-by-value. In this kind of situation, when a function wants to pass information **back** to the calling function, we must use our second parameter passing mechanism, **pass-by-reference**. Syntactically, this is really simple. We simply add an & symbol in the function's parameter list, like this:

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

We will discuss the difference between pass-by-value and pass-by-reference a lot more in the next lesson. For now you just need to know that when information is being passed from a function **back** to the function that called it, put & in front of the parameter. Here is our final version of drawBox. I promise we will never change it again!

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

```
void drawHorizontalLine(int numXs);
void draw2VerticalLines(int numSpaces, int numRows);
void drawOneRow(int numSpaces);
void getDimensions(int &width, int &height);
void drawBox(int width, int height);

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;
}
```

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

```
void drawHorizontalLine(int numXs)
{
    int count;
    for (count = 0; count < numXs; count++){
        cout << "X";
    }
    cout << endl;
}
```

```
void draw2VerticalLines(int numSpaces, int numRows)
{
    int rowCount;

    for (rowCount = 0; rowCount < numRows; rowCount++){
        drawOneRow(numSpaces);
    }
}
```

```
void drawOneRow(int numSpaces)
{
    int spaceCount;

    cout << "X";
    for (spaceCount = 0; spaceCount < numSpaces; spaceCount++){
        cout << " ";
    }
    cout << "X" << endl;
}
```