19. Functions

Objectives

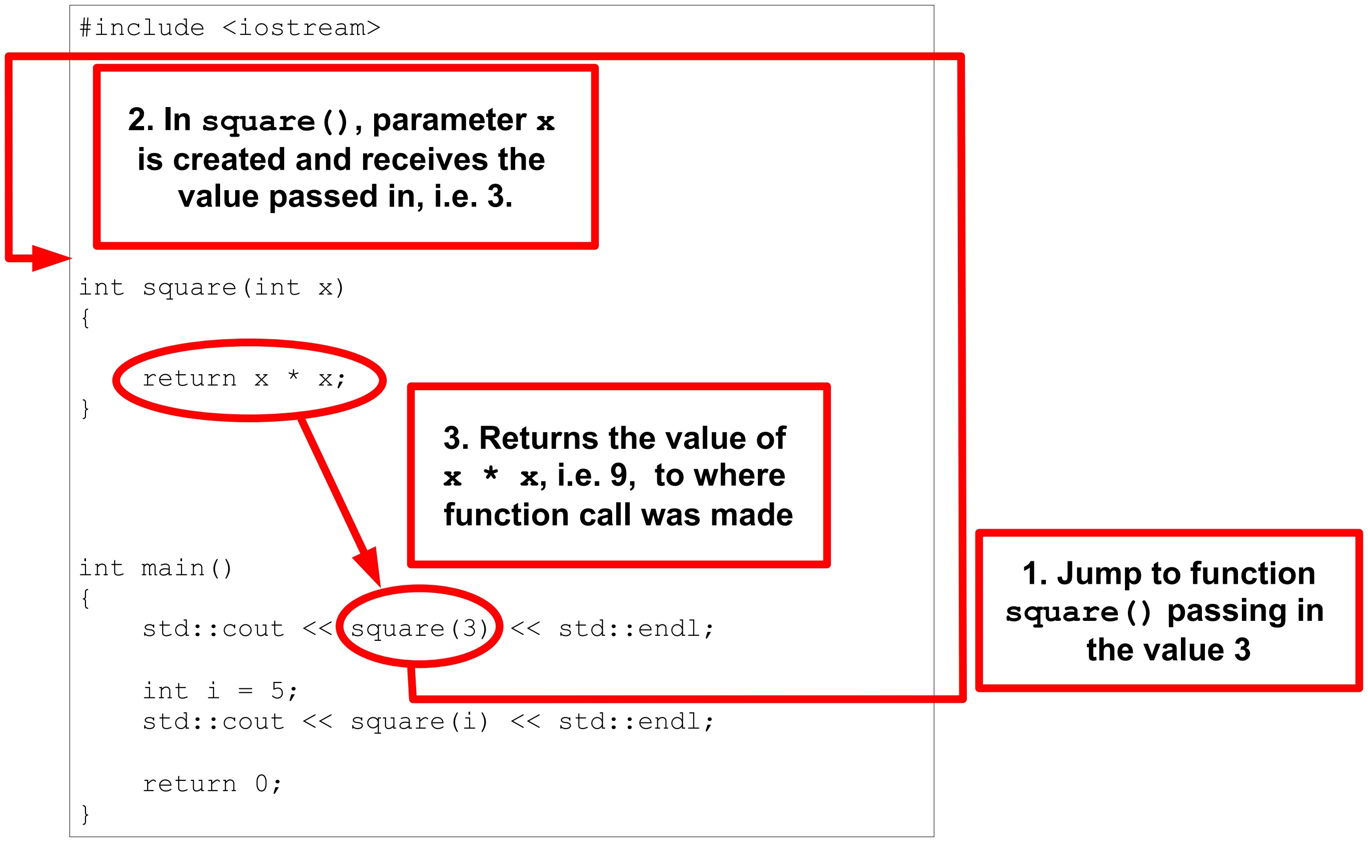
- Write functions with any number of parameters of basic type and either with or without a return value of basic type
- Understand pass-by-value
- Understand function scope
- Understand global scope
- Understand the negative impact of global variables
- Use global constants
- Understand pass-by-reference
- Use pass-by-function when appropriate
- Understand function prototypes
- Write function prototypes

Functions

OK. Here's a program. Run it. Can you guess what's happening?

```
#include <iostream>

int square(int x)
{
    return x * x;
}

int main()
{
    std::cout << square(3) << std::endl;

    int i = 5;
    std::cout << square(i) << std::endl;

    return 0;
}
```

First let me talk about the flow of execution of calling the function:



```
#include <iostream>

int square(int x)
{
    return x * x;
}

int main()
{
    std::cout << square(3) << std::endl;

    int i = 5;
    std::cout << square(i) << std::endl;

    return 0;
}
```

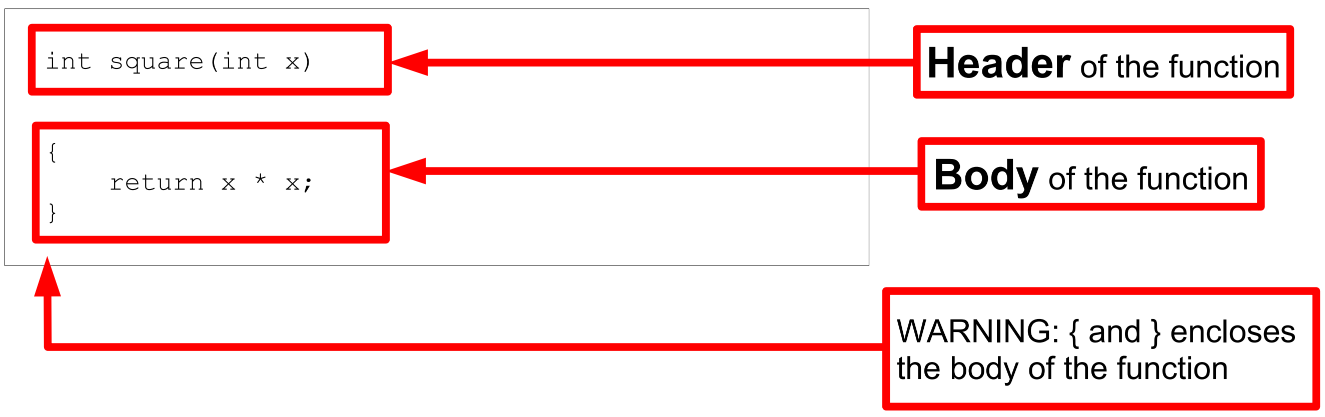
2. In square(), parameter x is created and receives the value passed in, i.e. 3.
3. Returns the value of x * x, i.e. 9, to where function call was made
1. Jump to function square() passing in the value 3

Now let's go over some terms and the syntax of a function.

This is called a **function**:

```
int square(int x)
{
    return x * x;
}
```

The **name** of the function is `square()`.

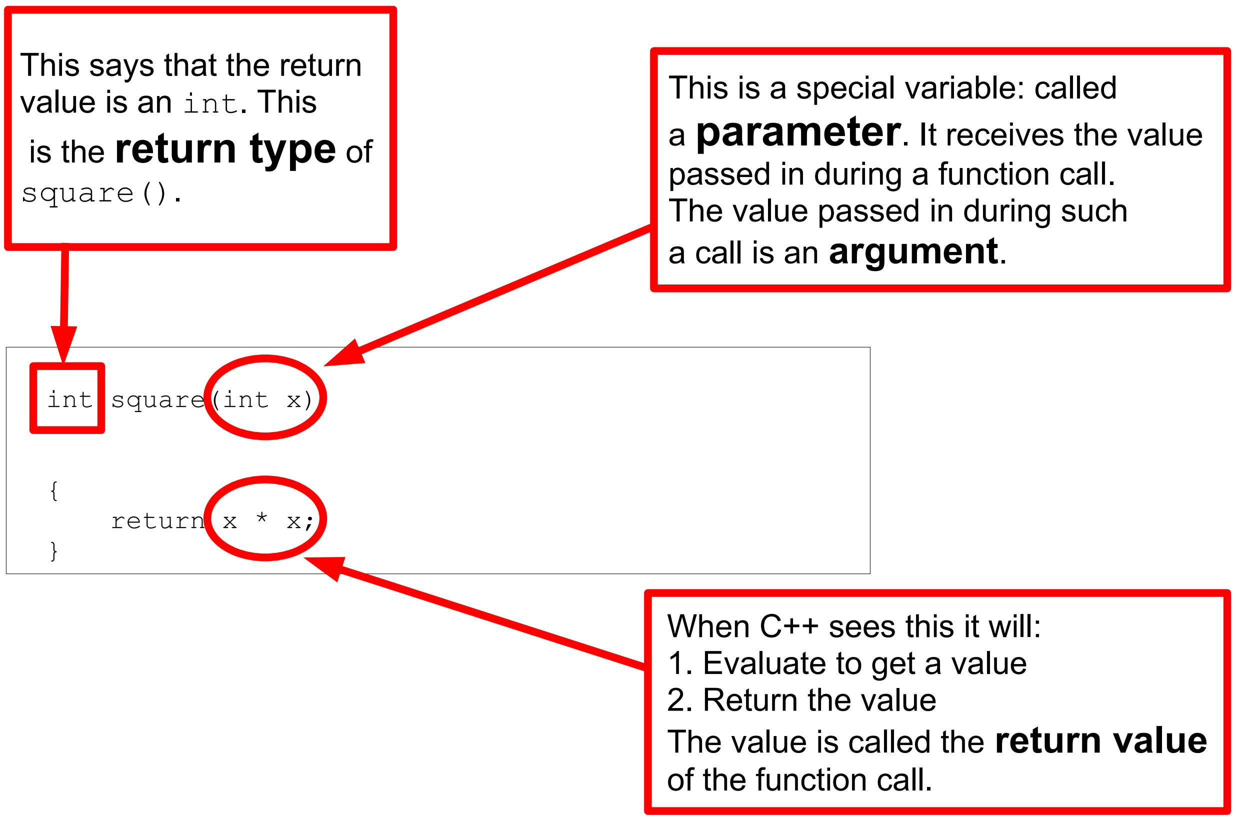


```
int square(int x)
{
    return x * x;
}
```

Header of the function

Body of the function

WARNING: { and } encloses the body of the function



This says that the return value is an `int`. This is the **return type** of `square()`.

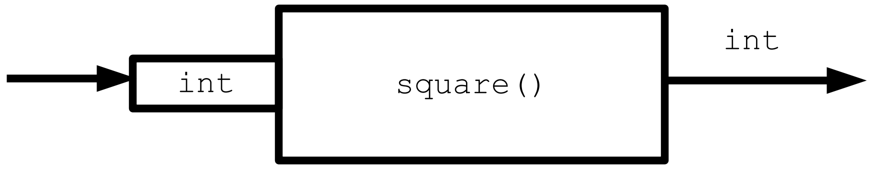
This is a special variable: called a **parameter**. It receives the value passed in during a function call. The value passed in during such a call is an **argument**.

```
int square(int x)
{
    return x * x;
}
```

When C++ sees this it will:
1. Evaluate to get a value
2. Return the value

The value is called the **return value** of the function call.

If you like you can think of the above function as a machine that takes a value, give the value to x, compute and return a value:



Note that the body of the function is just like the code you write in `main()`. (In fact `main()` ***is*** a function!) You can have as many statements as you like as long as they are valid statements.

Let's get C++ to show us the flow of execution into and out of the function. Modify the previous program and run it:

```
#include <iostream>

int square(int x)
{
    std::cout << "in square() ... "
              << "x = " << x << std::endl;
    int y = x * x;
    std::cout << "exiting square() ... "
              << "returning " << y << std::endl;
    return y;
}


int main()
{
    std::cout << "calling square() with 3 ...";
    int a = square(3);
    std::cout << "... back from square() ... ";
              << "received " << a << std::endl;

    int i = 5;
    std::cout << "calling square() with 5 ...";
    std::cout << square(i) << std::endl;
    a = square(i);
    std::cout << "... back from square() ... ";
              << "received " << a << std::endl;

    return 0;
}
```

In our example, the function `main()` calls the function `square()`. We say that `main()` is the **caller** and `square()` is the **callee**.
The following is the format for creating a function:

[return type] [func name]([param type] [var name])
{
        [body of function]
}

Exercise. Using the above program, add a `cube()` function that accepts an integer value and returns the cube of that value. In your `main()`, write a `for`-loop to print the cubes of 0, 1, 2, ..., 10 using the `cube()` function. (Use a `for`-loop, correct?) Run your program and verify it works correctly.

Exercise. Using the above program, add a `reciprocal()` function that accepts a `double` value and returns the reciprocal of that value. In your `main()`, write a `for`-loop to print `reciprocal(1)`, `reciprocal(2)`, ..., `reciprocal(10)`. Note in this case that the parameter (variable that receives a value) is a `double`. The return value is also a `double`. Therefore your function should look like this:

```
...

double reciprocal(double x)
{
    ...
}


int main()
{
    std::cout << reciprocal(1) << '\n';
    std::cout << reciprocal(2) << '\n';
    std::cout << reciprocal(10) << '\n';

    return 0;
}
```

Run your program and verify it works correctly.

Exercise. Write a function `sign()` that accepts a `double` and returns 1 if the `double` is > 0, 0 if the `double` is 0, or -1 if the `double` is < 0. Note in this case that the return value is an `int` (yes you can use a `double` but that would be an overkill – not to mention waste of memory since a `double` takes up more space.) Here's skeleton for you:

```
int sign(double x)
{
    ...
}
```

Of course you should always test your code.

Exercise. Write a function `sum()` that accepts an integer value and returns the sum of all integer values from 0 to the given integer value. For instance `sum(0)` returns 0; `sum(3)` returns 6 (since 0 + 1 + 2 + 3 gives 6). Here's something that might help:

```
#include <iostream>

int sum(int n)
{
    int s = 0;
    // write a for-loop so that s = 0 + 1 + ... + n
    return s;
}

int main()
{
    for (int i = 0; 5; ++i)
    {
        std::cout << sum(i) << '\n';
    }

    return 0;
}
```

Exercise. Write a function `factorial()` that accepts an integer value and returns the factorial of that integer value. Make sure you test your function.

Pass-by-Value

It's extremely important to remember that `main()` calls `square()` by passing in a value. This is called **pass-by-value**. The parameter (i.e. `x`) receives the value.

Here's a **very common gotcha**: If `main()` calls `square()` with a variable say `i`, and `x` (in `square()`) receives the **value** of `i`, then modifying `x` (in `square()`) does NOT affect the value of `i`.

Here's an example where the value of `i` is passed in to our `square()` function with `x` receiving the value. We change the value of `x` and back in `main()`, then value of `i` is not affected.

```
#include <iostream>

int square(int x)
{
    int y = x * x;
    x = 0;
    return x * x;
}


int main()
{
    int i = 5;
    std::cout << square(i) << std::endl;
    std::cout << i << std::endl;

    return 0;
}
```

Many students think that somehow the **variable** `i` (in `main()`) is passed in and that `x` (in `square()`) is somehow tied to the i (in `main()`).

That's **NOT TRUE**. Only the value of `i` (in `main()`) is passed onto the `x` (in `square()`). **The two variables have their own memory spaces. Changing one does not change the other.**

It's also important to remember that the memory of `square()` is created when you call it. When you return to `main()` (i.e. when you exit the block for `square()`), the memory for that particular call of `square()` is destroyed.

The following example shows you that when you call `square()`, the memory created for that call is different from the memory of `main()`. We'll create a variable `i` in `square()` and set it to zero. The `i` (in `main()`) is not changed.

```
#include <iostream>

int square(int x)
{
    int y = x * x;
    int i = 0;
    return x * x;
}


int main()
{
    int i = 5;
    std::cout << square(i) << std::endl;
    std::cout << i << std::endl;

    return 0;
}
```

A mental computational model

To explain the execution and return of a function call, here's a model of our "computer".

We're adding a special container, called the **stack**, that will allow us to do three things:

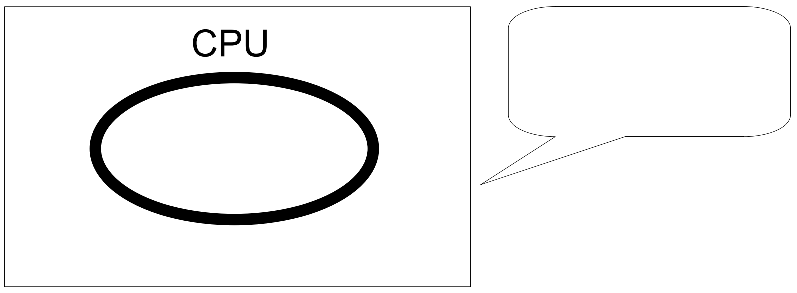
- Remember the **point of return**
- Put **value(s) to pass to** the function
- Put **value to pass back** from the function

Remember that **this is only a model**.

Here's the program we will execute on our model. (We won't talk about #include. That will come soon.)

```
#include <iostream>

int square(int x)
{
  return x * x;
}

int main()
{
    int i = 5;
    int j = square(i + 1);
    std::cout << j << std::endl;

    return 0;
}
```

So here's your computer again:

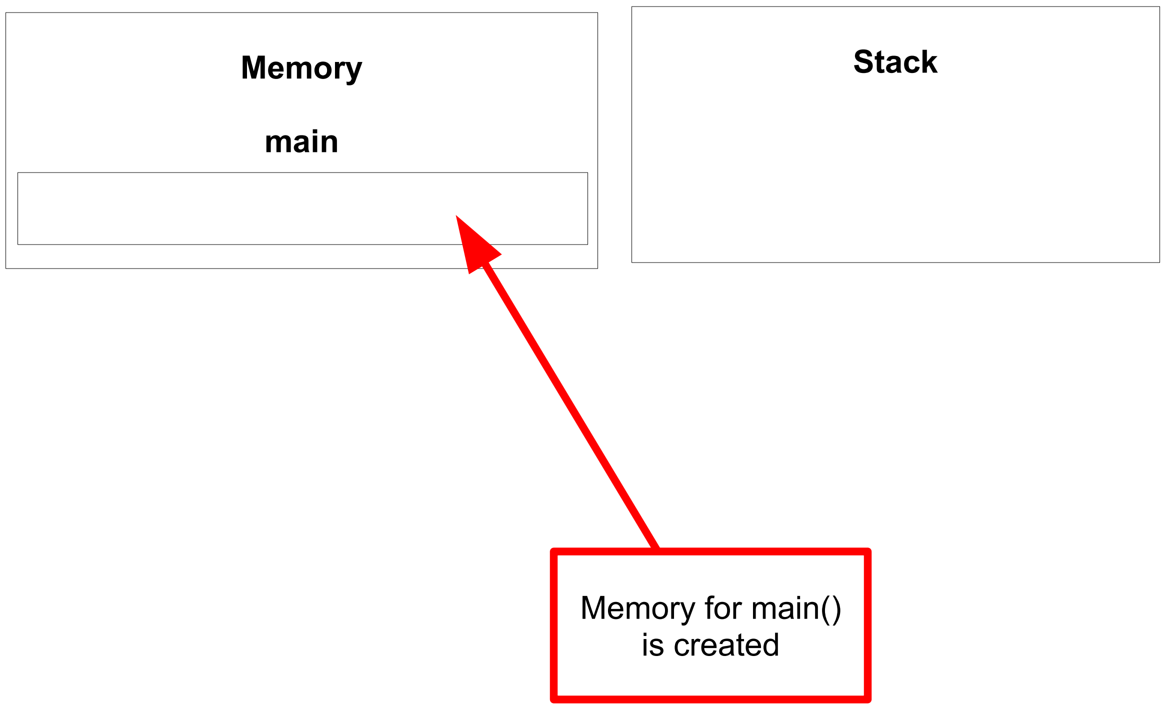


As before the speech bubble is the output. Everything printed appears in it.

Now when you run the program, the computer loads the program:

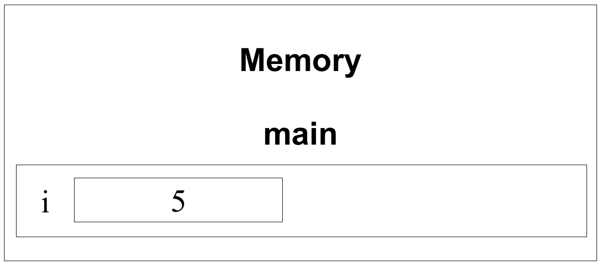
Code

```
#include <iostream>

int square(int x)
{
  return x*x;
}

int main()
{
    int i = 5;
    int j = square(i + 1);
    std::cout << j << std::endl;

    return 0;
}
```

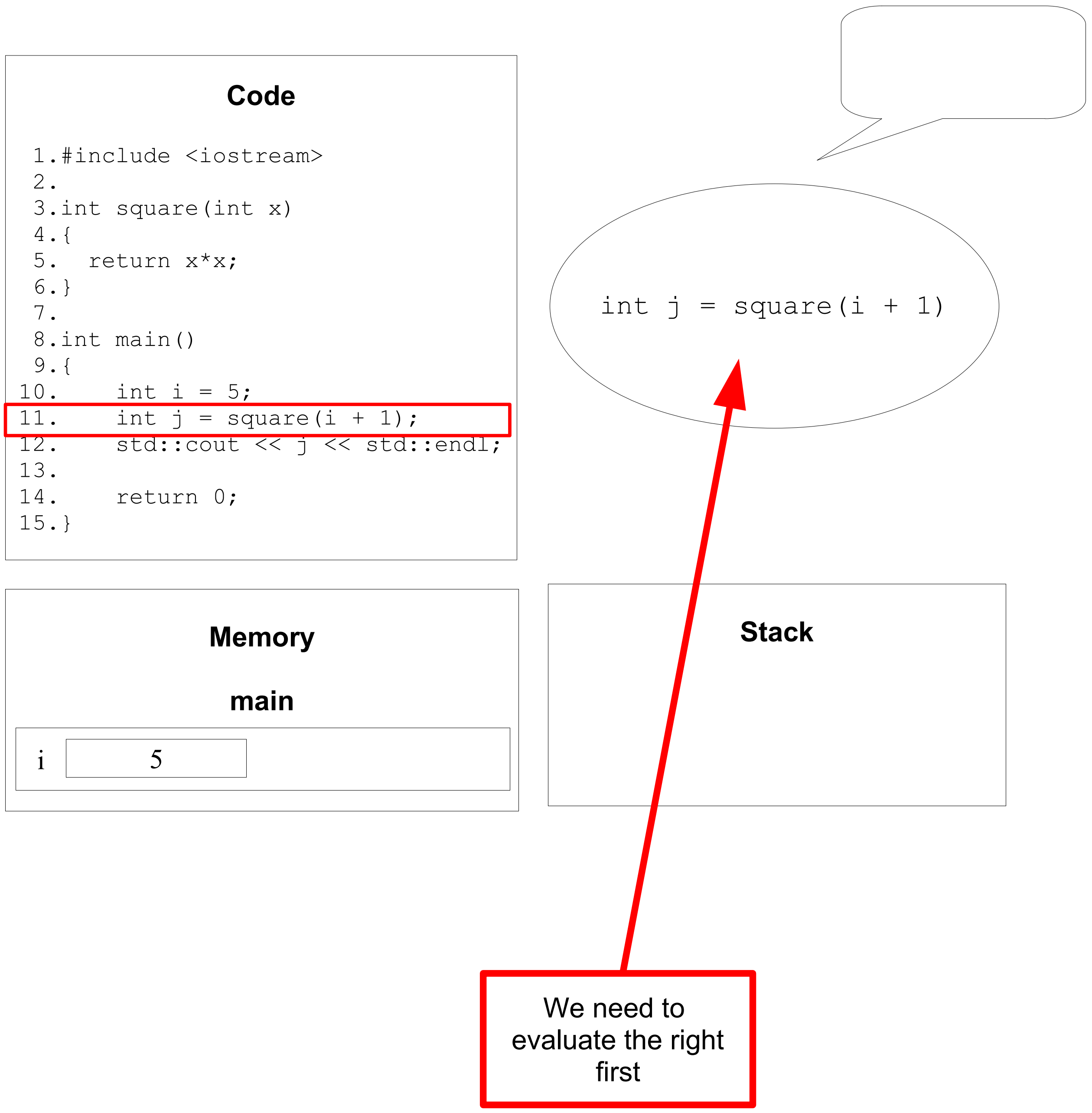


Code

```
#include <iostream>

int square(int x)
{
  return x*x;
}

int main()
{
    int i = 5;
    int j = square(i + 1);
    std::cout << j << std::endl;

    return 0;
}
```



Stack



Code

```
#include <iostream>

int square(int x)
{
  return x*x;
}

int main()
{
    int i = 5;
    int j = square(i + 1);
    std::cout << j << std::endl;

    return 0;
}
```

Code

```
#include <iostream>

int square(int x)
{
   return x*x;
}

int main()
{
    int i = 5;
    int j = square(i + 1);
    std::cout << j << std::endl;

    return 0;
}
```

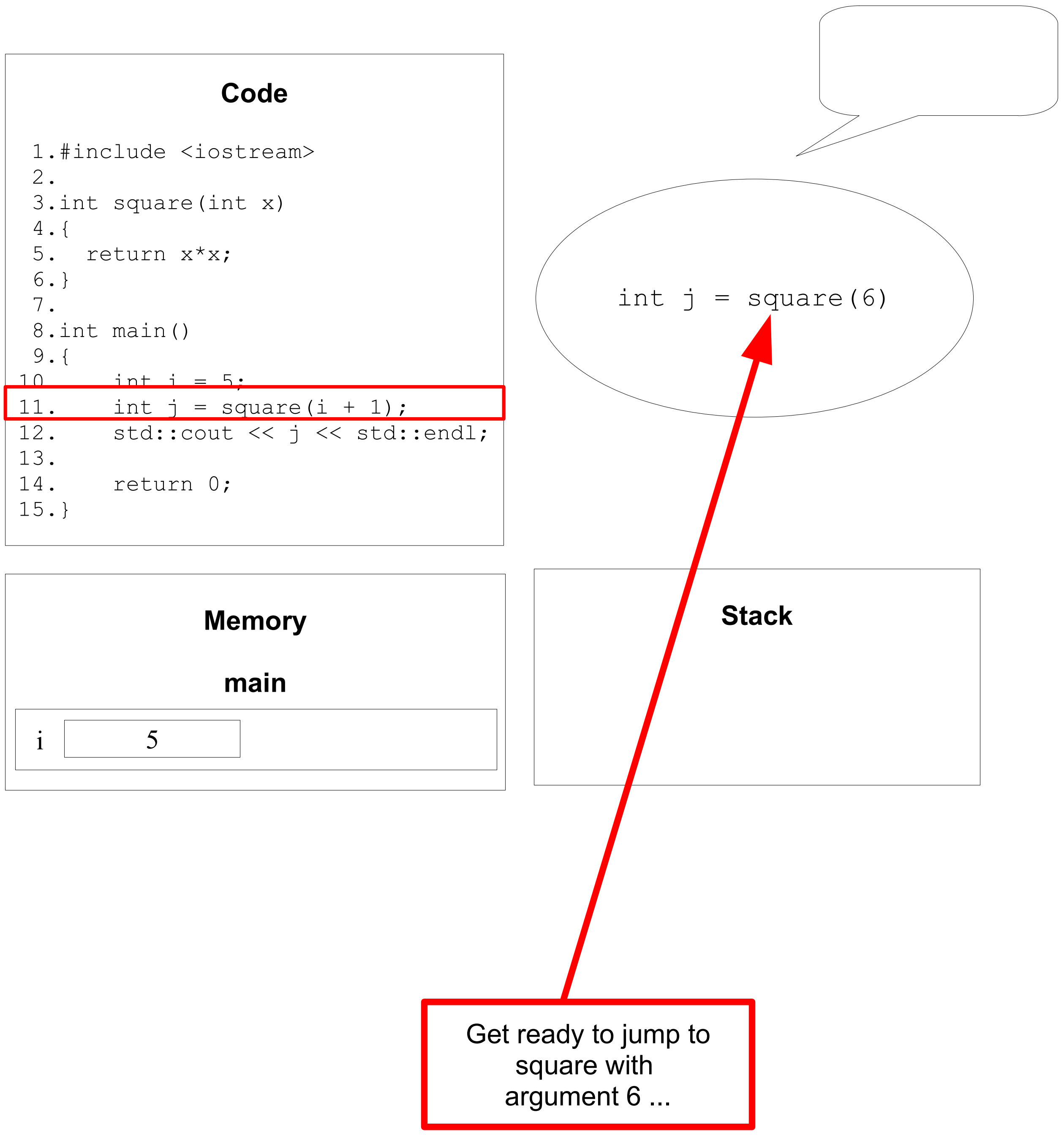
```
int j = square(5 + 1)
```

Memory

main

i	5

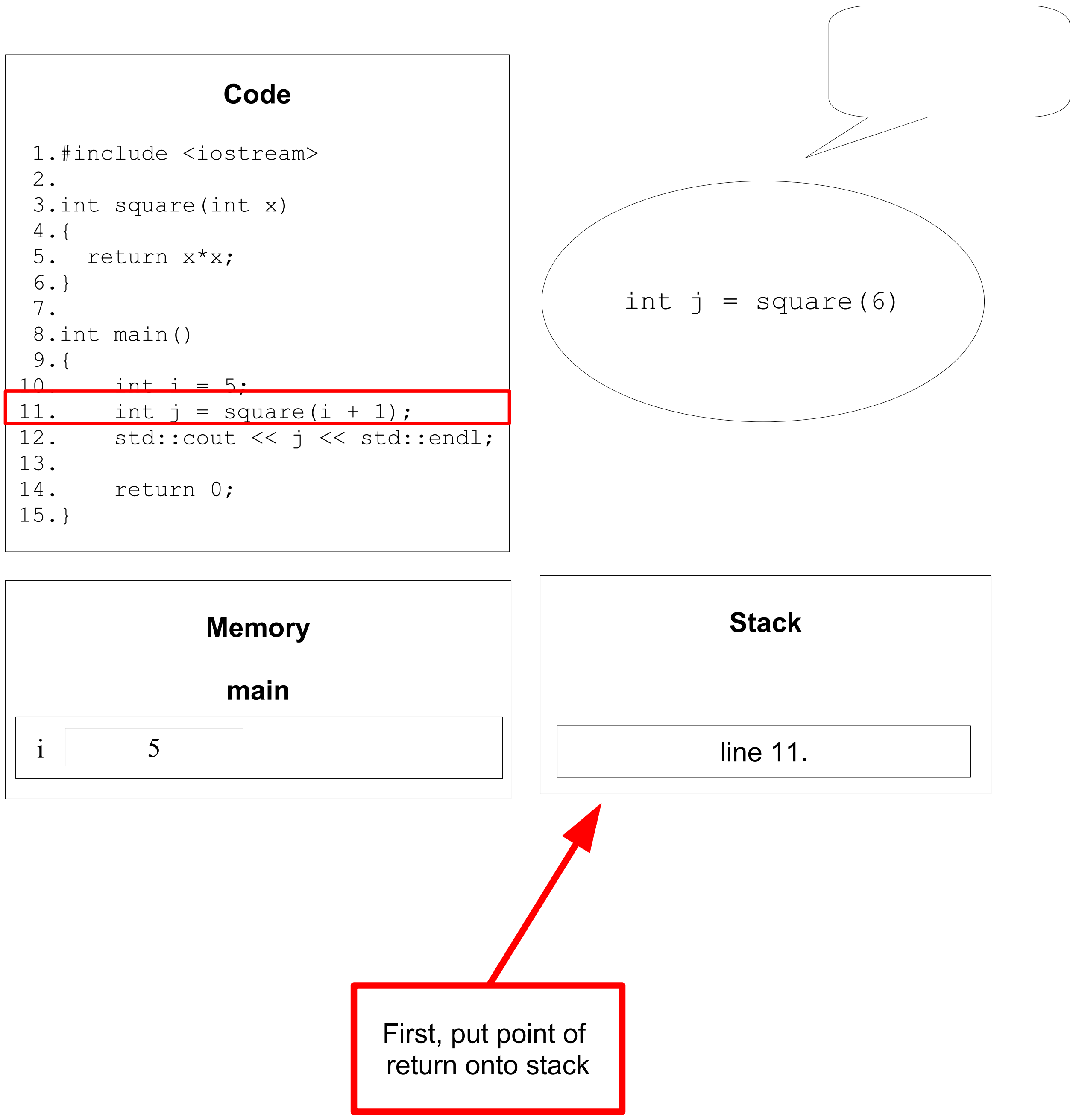
Stack



Code

```
#include <iostream>

int square(int x)
{
  return x*x;
}

int main()
{
    int i = 5;
    int j = square(i + 1);
    std::cout << j << std::endl;

    return 0;
}
```

Get ready to jump to square with argument 6 ...



Code

```
1.#include <iostream>
2.
3.int square(int x)
4.{
5.   return x*x;
6.}
7.
8.int main()
9.{
10.     int i = 5;
11.     int j = square(i + 1);
12.     std::cout << j << std::endl;
13.
14.     return 0;
15.}
```

```
int j = square(6)
```

Memory

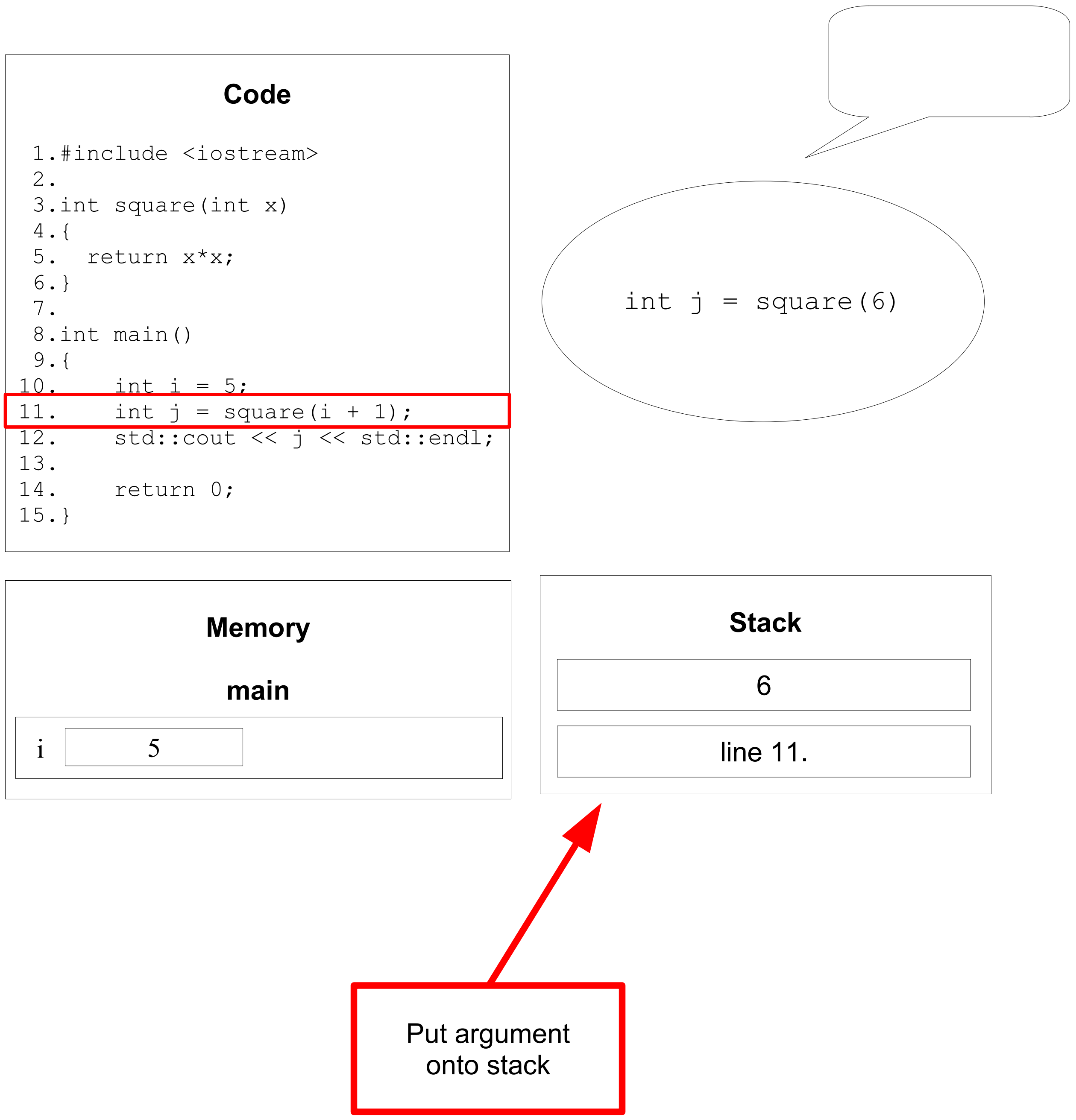
main

i	5

Stack

line 11.

First, put point of return onto stack

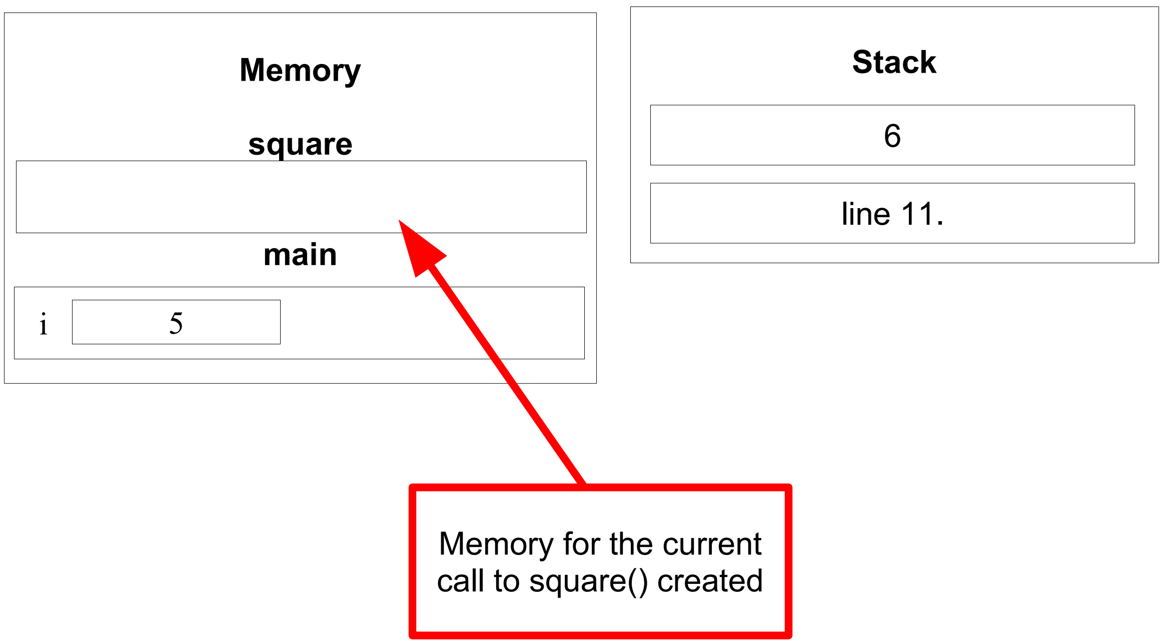


Code

```
1.#include <iostream>
2.
3.int square(int x)
4.{
5.   return x*x;
6.}
7.
8.int main()
9.{
10.     int i = 5;
11.     int j = square(i + 1);
12.     std::cout << j << std::endl;
13.
14.     return 0;
15.}
```

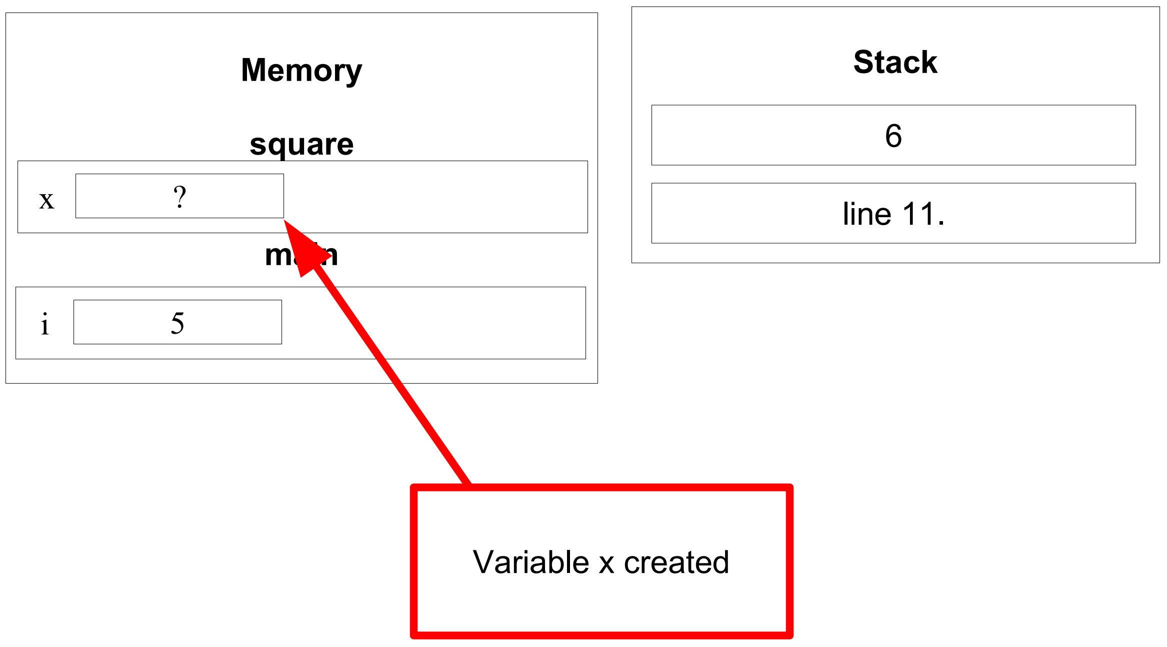
Code

```
 1.#include <iostream>
 2.
 3.int square(int x)
 4.{
 5.  return x*x;
 6.}
 7.
 8.int main()
 9.{
10.    int i = 5;
11.    int j = square(i + 1);
12.    std::cout << j << std::endl;
13.
14.    return 0;
15.}
```



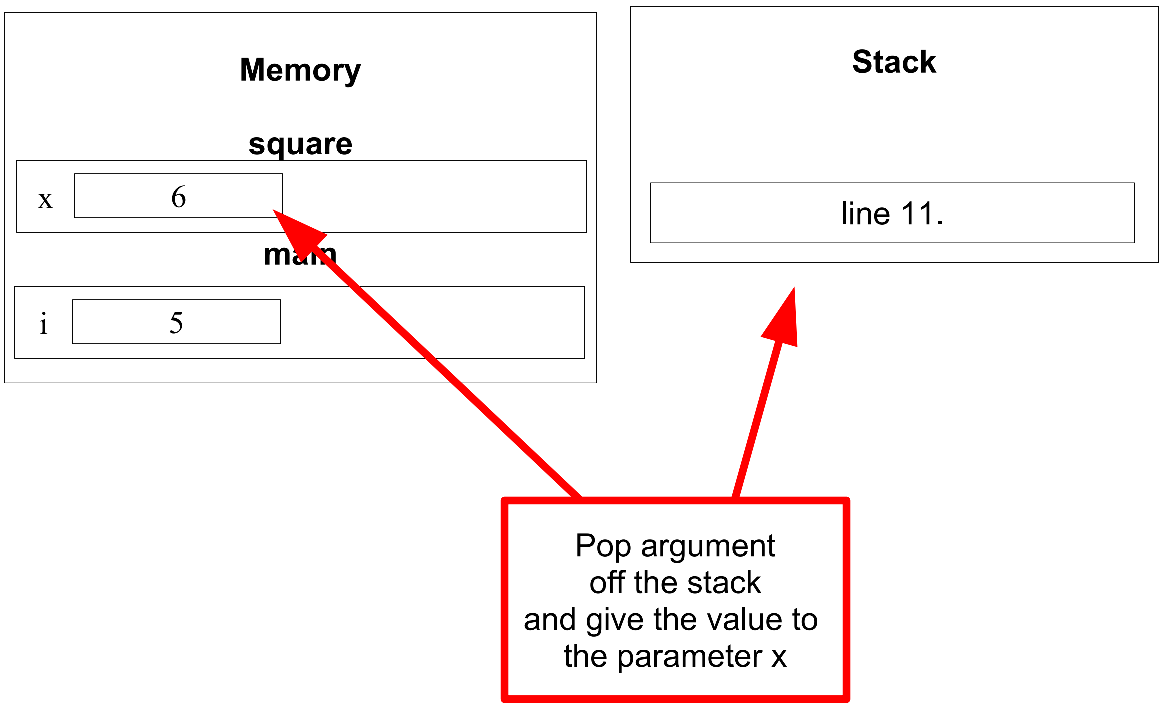
Code

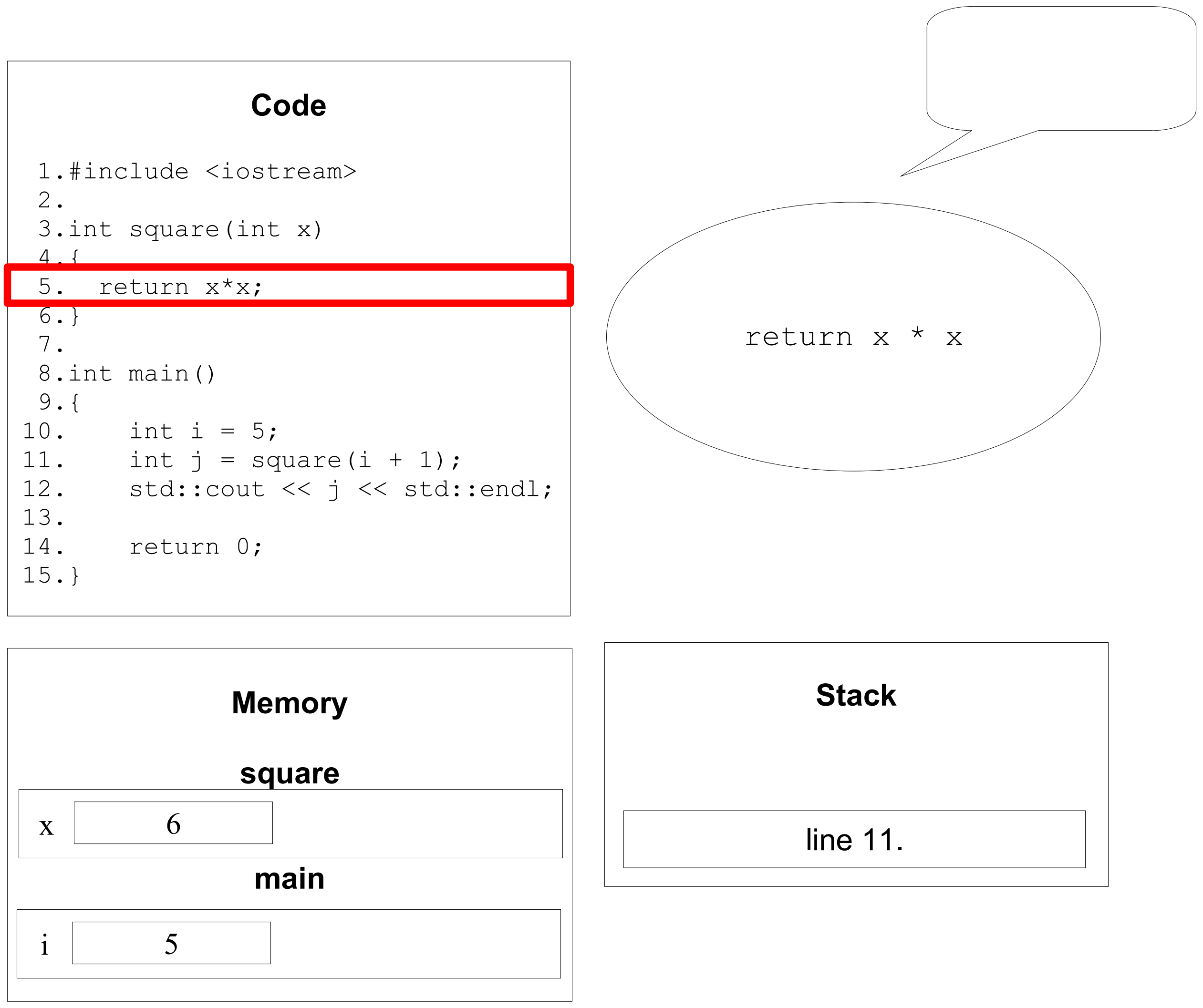
```
1.#include <iostream>
2.
3.int square(int x)
4.{
5.   return x*x;
6.}
7.
8.int main()
9.{
10.     int i = 5;
11.     int j = square(i + 1);
12.     std::cout << j << std::endl;
13.
14.     return 0;
15.}
```



Code

```
 1.#include <iostream>
 2.
 3.int square(int x)
 4.{
 5.  return x*x;
 6.}
 7.
 8.int main()
 9.{
10.    int i = 5;
11.    int j = square(i + 1);
12.    std::cout << j << std::endl;
13.
14.    return 0;
15.}
```


Memory
square
x
6
main
i
5
Stack
line 11.
Pop argument off the stack and give the value to the parameter x



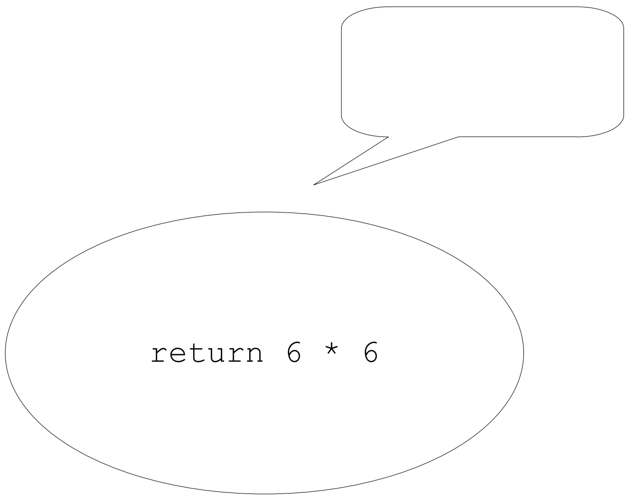
Code

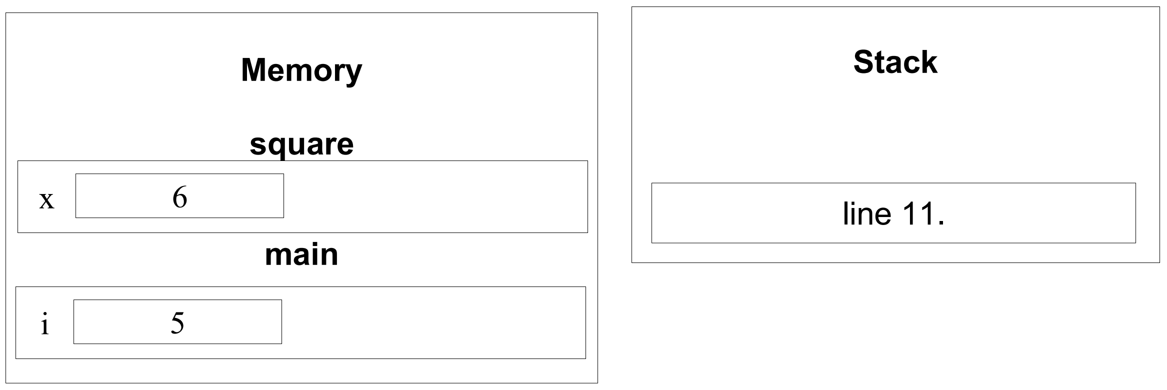
```
 1.#include <iostream>
 2.
 3.int square(int x)
 4.{
 5.  return x*x;
 6.}
 7.
 8.int main()
 9.{
10.    int i = 5;
11.    int j = square(i + 1);
12.    std::cout << j << std::endl;
13.
14.    return 0;
15.}
```

`return x * x`

Code

```
 1.#include <iostream>
 2.
 3.int square(int x)
 4.{
 5.  return x*x;
 6.}
 7.
 8.int main()
 9.{
10.    int i = 5;
11.    int j = square(i + 1);
12.    std::cout << j << std::endl;
13.
14.    return 0;
15.}
```





Code

```
 1.#include <iostream>
 2.
 3.int square(int x)
 4.{
 5.  return x*x;
 6.}
 7.
 8.int main()
 9.{
10.    int i = 5;
11.    int j = square(i + 1);
12.    std::cout << j << std::endl;
13.
14.    return 0;
15.}
```

`return 36`

Memory

square

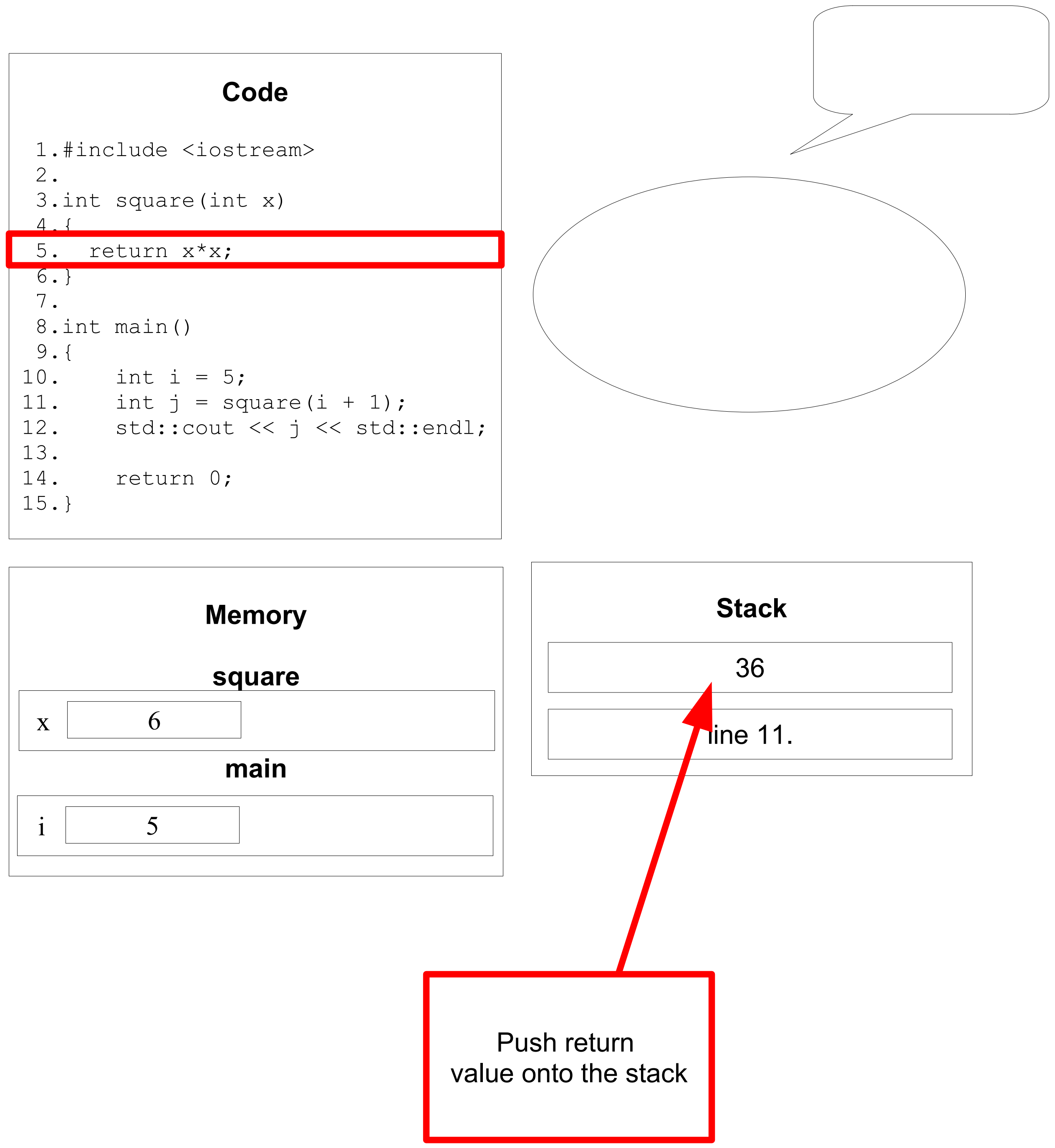
x	6

main

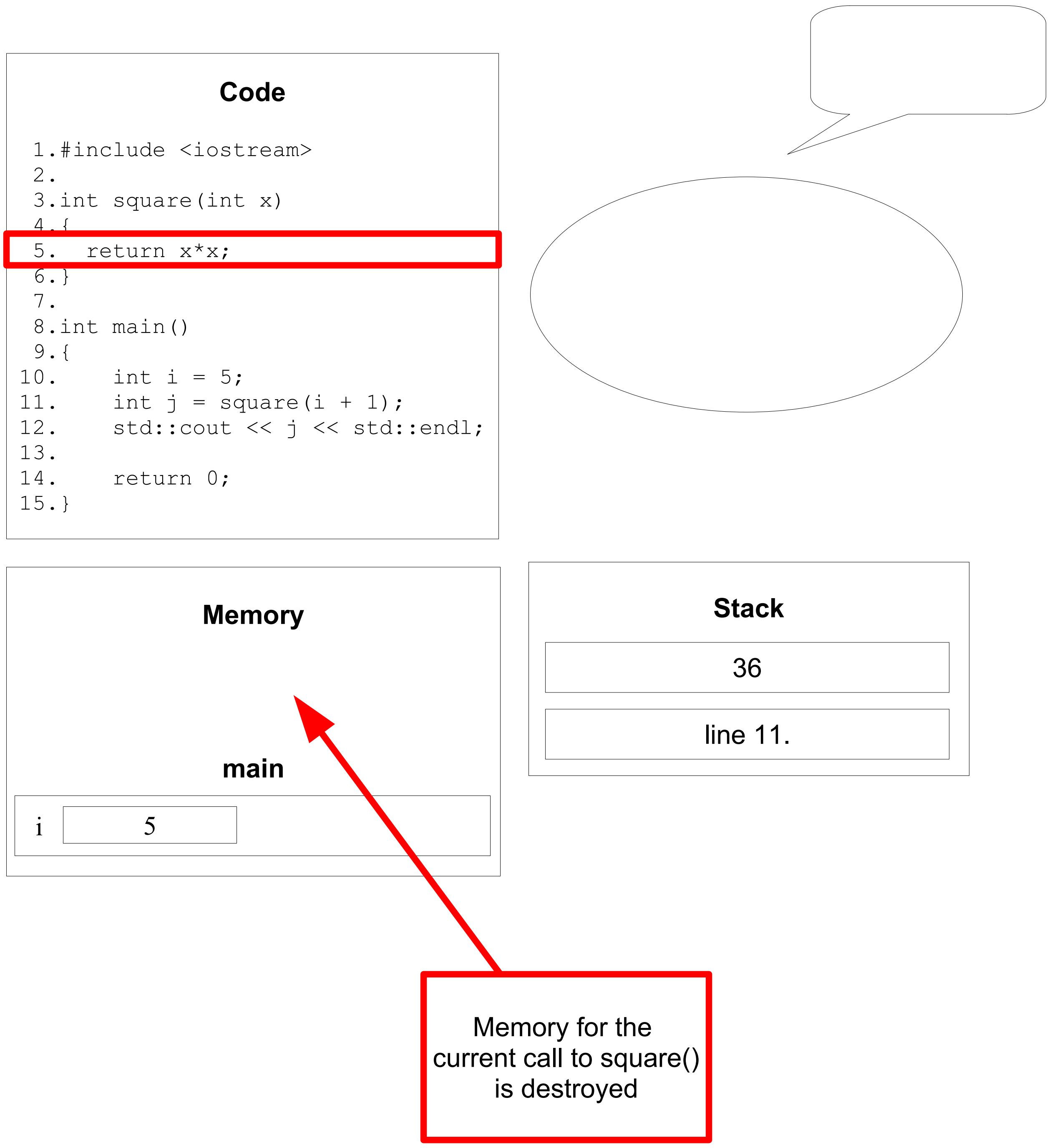
i	5

Stack

line 11.

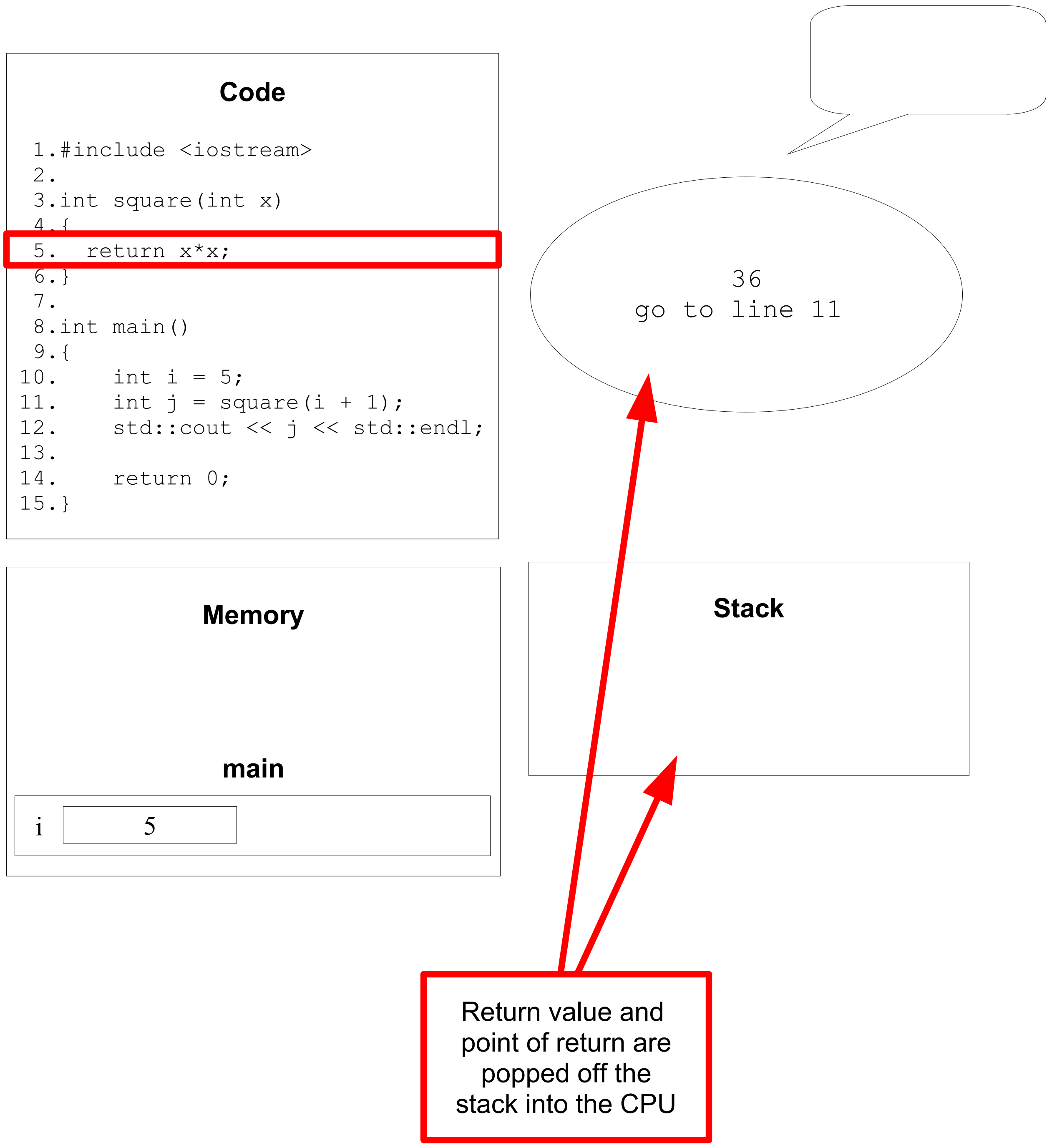


```
 1.#include <iostream>
 2.
 3.int square(int x)
 4.{
 5.   return x*x;
 6.}
 7.
 8.int main()
 9.{
10.     int i = 5;
11.     int j = square(i + 1);
12.     std::cout << j << std::endl;
13.
14.     return 0;
15.}
```



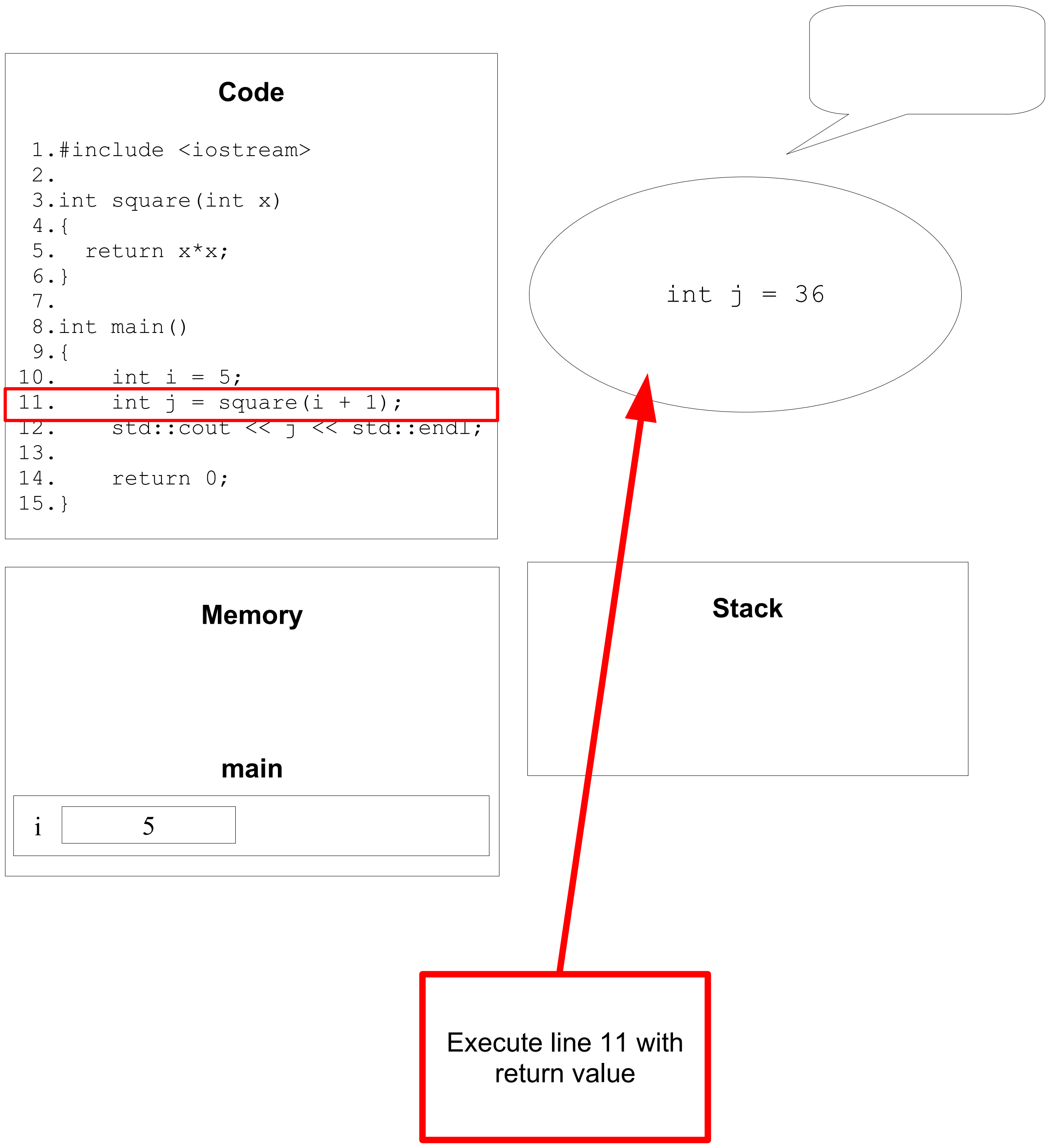
Code

```
 1.#include <iostream>
 2.
 3.int square(int x)
 4.{
 5.  return x*x;
 6.}
 7.
 8.int main()
 9.{
10.    int i = 5;
11.    int j = square(i + 1);
12.    std::cout << j << std::endl;
13.
14.    return 0;
15.}
```



```
 1.#include <iostream>
 2.
 3.int square(int x)
 4.{
 5.  return x*x;
 6.}
 7.
 8.int main()
 9.{
10.    int i = 5;
11.    int j = square(i + 1);
12.    std::cout << j << std::endl;
13.
14.    return 0;
15.}
```

Return value and point of return are popped off the stack into the CPU

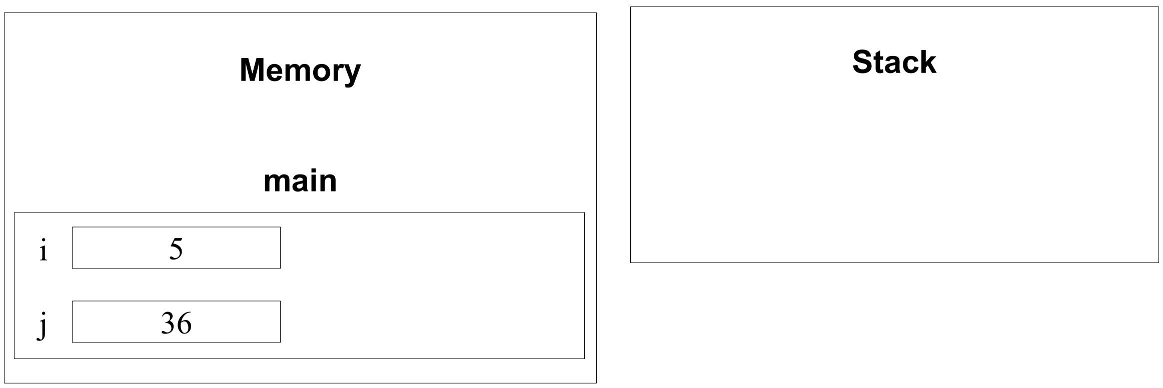


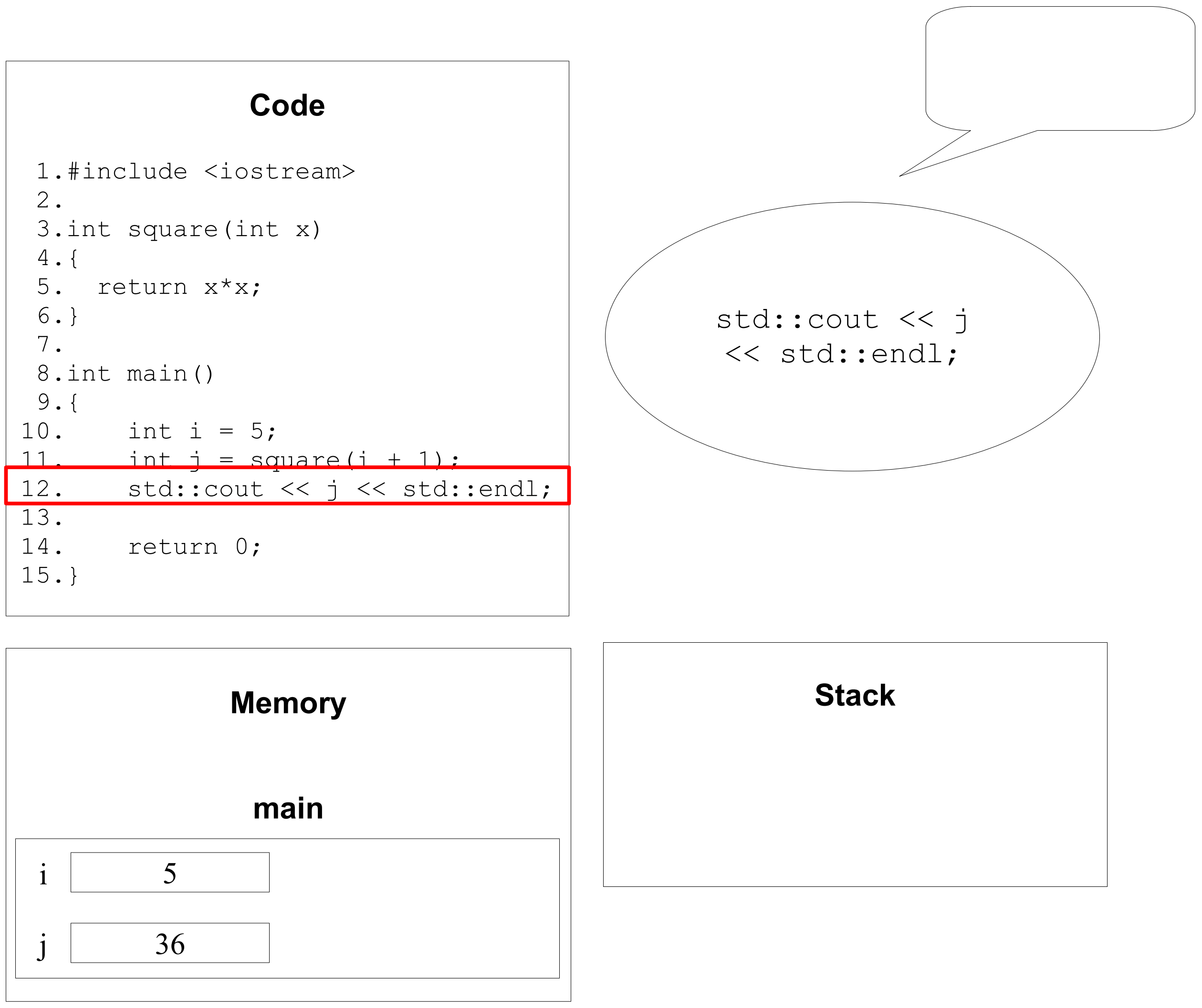
Code

```
 1.#include <iostream>
 2.
 3.int square(int x)
 4.{
 5.   return x*x;
 6.}
 7.
 8.int main()
 9.{
10.    int i = 5;
11.    int j = square(i + 1);
12.    std::cout << j << std::endl;
13.
14.    return 0;
15.}
```

Code

```
#include <iostream>

int square(int x)
{
  return x*x;
}

int main()
{
    int i = 5;
    int j = square(i + 1);
    std::cout << j << std::endl;

    return 0;
}
```



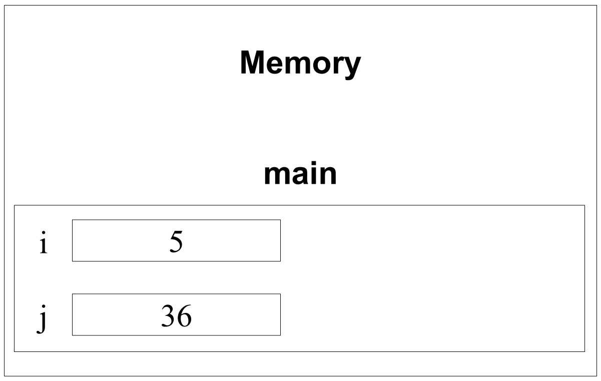


```
#include <iostream>

int square(int x)
{
  return x*x;
}

int main()
{
    int i = 5;
    int j = square(i + 1);
    std::cout << j << std::endl;

    return 0;
}
```

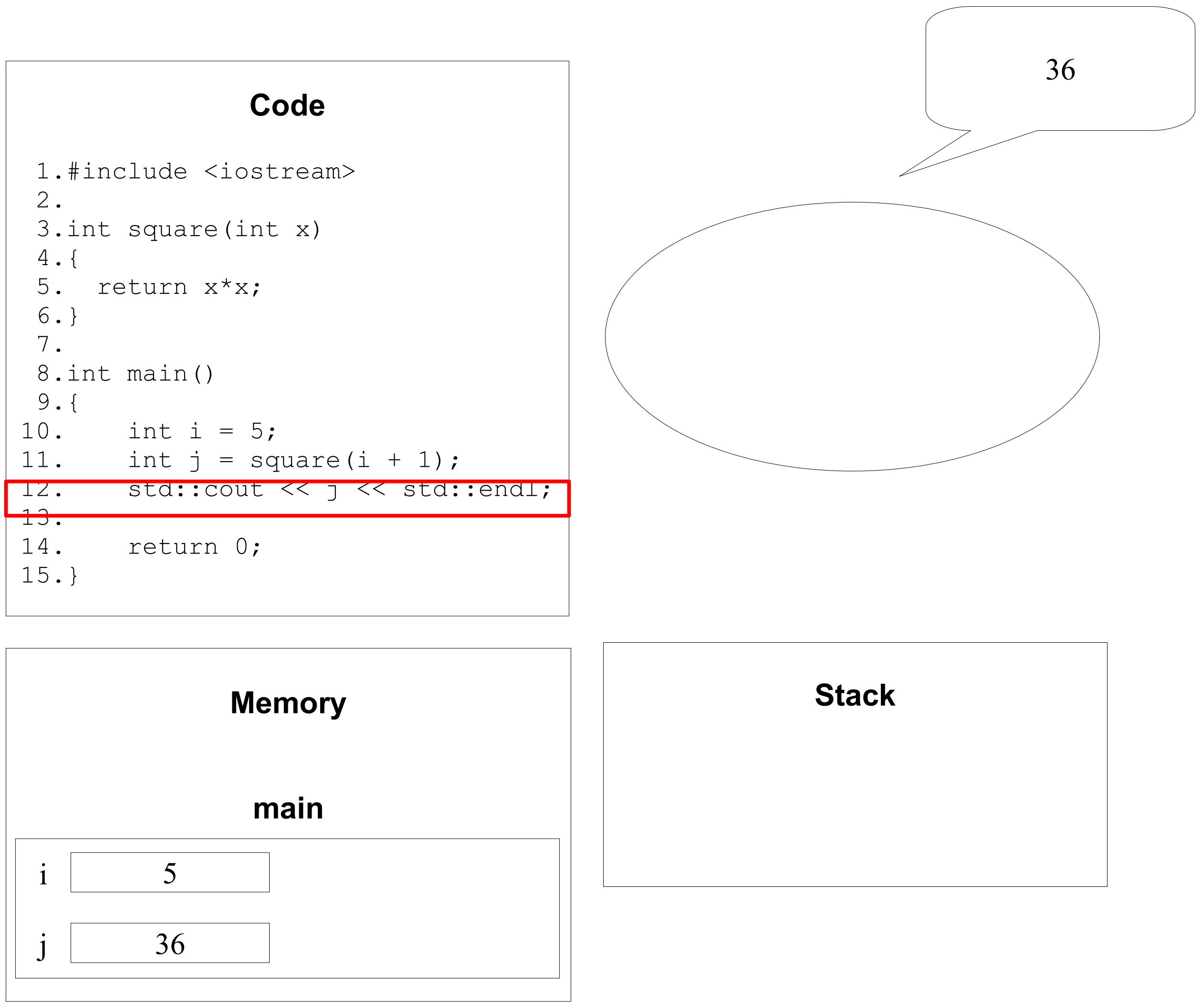
Code

```
#include <iostream>

int square(int x)
{
  return x*x;
}

int main()
{
    int i = 5;
    int j = square(i + 1);
    std::cout << j << std::endl;

    return 0;
}
```

```
std::cout << 36
 << std::endl;
```



Stack



```
#include <iostream>

int square(int x)
{
  return x*x;
}

int main()
{
    int i = 5;
    int j = square(i + 1);
    std::cout << j << std::endl;

    return 0;
}
```

That's it!

There are several important things to observe in this simulation:

- The memory for the call to `square()` is different from the memory of `main()`
- Values are passed between `main()` and `square()` and not the variables.

By the way there's something special about the stack. A **stack** is a data structure (i.e. container of data with operations) that allows you to put things into it (this operation is called **push**) and also to get something out of it (the operation is called **pop**). You cannot choose what you want to get out of the stack: It will always be the last thing that you put into it. That's why the stack is also called a **last-in-first-**

out (LIFO) data structure.

A stack of plates at a buffet is (almost) a stack. The plate you take (if you only take from the top) is the last plate placed onto the stack.

The above model illustrates many concepts in function calling. But it's only a model. For the real picture you have to take CISS360 (Assembly Language and Computer Systems) and to understand how Physics (or electronics) can be used to physically implement a computational device you have to take CISS430 (Computer Architecture).

By the way, it's actually **OK not to use a return value**. For instance the following program is valid:

```
int square(int x)
{
    return x * x;
}

int main()
{
    square(4);

    return 0;
}
```

As you can see in the above code the value 16 is neither being used in any computation or nor in a print statement. It's just thrown away.

In fact something like this is also valid! (Although quite silly).

```
int main()
{
    16;

    return 0;
}
```

Run it. Again the value 16 is not used in any way.

Note again that when C++ returned from `square()`, the memory used for `square()`'s variables is destroyed. This is similar to exiting for instance a for-loop. So in this code segment that uses `square()` twice:

```
...
std::cout << square(5) << std::endl;
...
std::cout << square(i) << std::endl;
```

the memory for `square()` to hold data is created and destroyed twice.

Exercise. Write a function `isprime()` that accepts an integer value and returns `true` if the integer value is a prime; otherwise `false` is returned. (Refer to previous notes on checking for primeness.) Test your function thoroughly. Note that the value passed into the function is an `int`. The value returned is a `bool`. Therefore the function looks like this:

```
bool isprime(int n)
{
    ...
}
```

In your main(), write a for-loop that prints all the primes up to 1000000000 using the above function.

Exercise. Using the above function, print all twin primes less than 1000000000, i.e. print p and p + 2 such that p and p+2 are primes and less than 100000000.

Returning nothing

You can write a function that does not return a value:

```
#include <iostream>

void printHelloWorld(int x)
{
    std::cout << "Hello, World!" << std::endl
              << "Function received: "
              << x
              << std::endl;
    return;
}


int main()
{
    printHelloWorld(3);

    return 0;
}
```

Question: What is the return type of a function that does not return a value? (Look at the code above.)

Exercise. Write a function `printYesNo()` that accepts a boolean value and prints yes if the value is `true` and no otherwise. Test your code. [Hint: You can use the `printHelloWorld()` function above. The parameter `x` is an `int`. You need to change it to a `bool`.]

Exercise. What's wrong here?

```
void f(int x)
{
    return x * x;
}

int main()
{
    int y = f(5);
    return 0;
}
```

Exercise. What is the output (without running the program)?

```
void increment(int x)
{
    x++;
    std::cout << x << std:::endl;
    return;
}

int main()
{
    int x = 1;
    increment(x);
    std::cout << x << std::endl;

    return 0;
}
```

Now verify by running the program. Correct the program using this:

```
____________ increment(int x)
{
    x++;
    return ____________;
}


int main()
{
    int x = 1;
    x = increment(x);
    std::cout << x << std::endl;
    return 0;
}
```

Exercise. Invent your own ASCII art problem (or take the code from earlier notes or assignments and write a function for it) and write a function for it. Here's one: Write the function `drawSquare()` that accepts an int value and draw a box with height and width of the specified `int` length. For instance calling `drawSquare(3)` prints this to the console window:

```
***
* *
***
```

and calling `drawSquare(8)` prints this:

```
********
*      *
*      *
*      *
*      *
*      *
*      *
********
```

Passing in nothing

You can also have a function that does not accept any value:

```
#include <iostream>

void printHelloWorld()
{
    std::cout << "hello, world!" << std::endl;
    return;
}


int main()
{
    printHelloWorld();
    return 0;
}
```

Of course you can have a function that accepts nothing but returns a value.

Exercise. The function `f()` accepts nothing and returns a `char`. Write the function header of `f()`:

```
________________________________
{
  ...
}
```

Exercise. The function `g()` accepts an `int` and returns a `double`. Write the function header of `g()`. (You can choose any name for the `int` parameter.)

```
________________________________
{
  ...
}
```

Exercise. The function `h()` accepts a `double` but does not return any value. Write the function header of `h()`. (You can choose any name for the `int` parameter.)

```
________________________________
{
  ...
}
```

Exercise. Write a function called `gimmeInt()` that does not have any parameter. Each time you call the function it returns the integer 42. Test your code with this

```
#include <iostream>

// put your function here

int main()
{
    for (int i = 0; i < 10; ++i)
    {
        std::cout << gimmeInt() << '\n';
    }

    return 0;
}
```

Scope

Functions are like variables: They have scopes too.

Exercise. Does this work?

```
#include <iostream>

int main()
{
    printHamAndEggs();
    return 0;
}


void printHamAndEggs()
{
    std::cout << "ham and eggs!";
    return;
}
```

Although you can nest for-loops to get double for-loops, you can nest if statements, you can nest while loops, you **cannot nest functions in C++**. Try this:

```
#include <iostream>

int main()
{
    void printHamAndEggs()
    {
         std::cout << "ham and eggs!";
         return;
    }

    printHamAndEggs();
    return 0;
}
```

BAD!!!

(Some programming languages do allow you to do this. But not C++).

Exercise. Will this work? Why?

```
#include <iostream>

void f()
{
     std::cout << matrixReloadedRating << "\n";
     return;
}


int main()
{
      int matrixReloadedRating = 1;
      f();
      return 0;
}
```

Verify with your compiler.

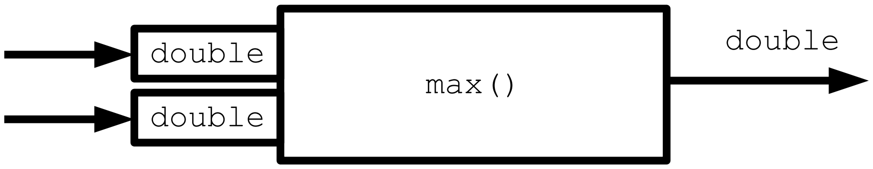
Functions with two parameters

I hope it's not too surprising that you can define a function with two parameters.

```
#include <iostream>


double max(double x, double y)
{
    if (x < y)
        return y;
    else
        return x;
}


int main()
{
    std::cout << max(5, 3) << std::endl;

    int x = 5, y = 3;
    std::cout << max(x * x, y + x) << std::endl;
    return 0;
}
```

You can have more than two return statements in a function.

Here's a picture that might help:



Note that in this case `max()` is called with values 25 and 8. The **ordering** of 25 and 8 is **important**!!! The x of max() will receive the first argument (i.e. 25) and y of max() will receive the second argument (i.e. 8).

You can think of a return statement as a point of exit from a function. Just like the for-loop and while-loop, you should NOT have too many points of exit because it makes you code much harder to trace and debug. For instance in the above `max()` function you can rewrite it as:

```
double max(double x, double y)
{
    int m = x;
    if (m < y) m = y;
    return m;
}
```

Another thing to note is that you are calling `max()` with two integers; `max()` expects two `double`s. That's the same as initializing the `x` and the `y` of `max()` with two integers; the integer values 25 and 8 are just promoted to `double`s 25.0 and 8.0. That's all there is to it. This is similar to declaring a `double` and initializing it with an integer:

```
double blah = 3;
```

It's clear that you can have a function of three parameters, four parameters, etc.

Note only that, the parameters need not have the same types. Something like this is OK:

```
void f(int x, double y, bool z, char c)
{
    ...
}
```

Exercise. Write a function `min()` that accepts two doubles and returns the minimum of their values. (What should the return type be?) Test your function thoroughly.

Exercise. Write a function `avg()` that accepts two doubles and returns their average. (What's the reasonable return type?) Test your code.

Exercise. Write a function `clip()` such that `clip(min, max, x)` (all values are doubles) will return x if min <= x and x <= max. Otherwise if x is greater than max, max is returned and if x is less than min, then min is returned. For instance `clip(0.0, 0.25, 1.0)` will return `0.25`, `clip(0.0, -1.2, 1.0)` will return `0.0`, and `clip(0.0, 5.25, 1.0)` returns `1.0`.

Exercise. Write a function `letterGrade()` that returns a letter grade given a student's percentage grade and the cutoff percentages for A, B, C, D. For instance this function call

```
letterGrade(86.5, 90.5, 79.6, 70, 60.2)
```

requests for the letter grade for the percentage grade 86.5 where a percentage grade of 90.5 or above is an `'A'`, 79.6 or above is a `'B'`, 70 or above is a `'C'`, 60.2 or above is a `'D'`, and in all other cases an `'F'` is returned.

Lots of exercises

Exercise. Write a function `printDollars()` that accepts an integer that represents the number of pennies and print the amount in dollars. For instance `printDollar(123)` will print

```
$1.23
```

What is the return type of this function?

Exercise. Write a function `randunit()` that returns a random double between 0.0 and 1.0. Verify your work.

Exercise. Write a function `randrange()` that accepts two integers a and b, and returns a random integer between a and b (inclusive). Verify your work.

Exercise. Write a function `apsum()` (sum of arithmetic progression) that accepts three doubles a, b, d and returns the sum of the terms a, a+d, a+2*d, ... which are at most b. (There are two cases: c is positive or c is negative.) Verify your work.

Exercise. Write a function `gpsum()` (sum of geometric progression) that accepts three `double`s a, b, r and returns the sum of the terms a, a*r, a*r*r, ... which are at most b. Verify your work.

Exercise. Write a function `getYOrN()` that continually prints the following message:

```
Yes or no (y/n)?
```

prompts the user for a character until either y or n is entered. Once either y or n is entered, that character is returned. Test it with the following `main()`:

```
#include <iostream>

// your function here
int main()
{
    std::cout << "do u need an extra head?\n";
    char answer = getYorN();

    if (answer == 'y')
    {
        std::cout << "step right up ...\n";
    }
    else
    {
        std::cout << "next ...\n";
    }

    return 0;
}
```

Exercise. Write a function `power()` so that `power(a, b)` returns `a` to the power of `b` where `a`, `b` are integers and `b >= 0`. (This is more or less a simplified `pow()` function.)

Functions calling functions

You can also have any function (other than `main()`) call another function:

```
#include <iostream>

void f()
{
    std::cout << "in f() ..." << std::endl;
    return;
}


void g()
{
    std::cout << "in g() ..." << std::endl;
    f();
    std::cout << "in g() again ..." << std::endl;
    return;
}


int main()
{
    g();
    return 0;
}
```

Exercise. Complete this code segment:

```
// Returns the square of n
int square(int n)
{
    ...
}

// Returns the sum of i*i for i = start, start+1,...,
// end
int sum_squares(int start, int end)
{
    ...
}
```

Write a `main()` to test `sum_squares()`.

Exercise. What is the output of this program:

```
#include <iostream>


int g(double x, double y)
{
    return x + y;
}


int f(int x, int y)
{
    return g(x, y) + x;
}


int main()
{
    int x = 5;
    double z = 3.14;

    std::cout << f(x, z) << std::endl;
    return 0;
}
```

Exercise. What is the output of this program:

```
#include <iostream>


int h(double x, int y)
{
    return x * y;
}


int g(double x, double y)
{
    return x + y;
}


int f(int x, int y)
{
    return g(x, y) + h(y, x);
}


int main()
{
    int x = 5, y = 7;
    double z = 3.14;

    std::cout << f(x, z) + g(y, x) << std::endl;
    return 0;
}
```

Exercise. Write a function `isleapyear()` such that `isleapyear(yyyy)` returns true exactly when the integer `yyyy` is a leap year. Make sure you test your function.

Exercise. Write a function `days_in_month()` such that `days_in_month(mm, yyyy)` returns the number of days for month `mm` in year `yyyy`. This function should use the `isleapyear()` function.

Exercise. Write a function `seconds_from_midnight()` such that `seconds_from_midnight(hhmmss)` returns the number of seconds since midnight for the integer 6-digit integer `hhmmss` which represents time in the hh:mm:ss 24-hour format. For instance `second_from_midnight(93045)` returns the number of seconds from midnight to 9:30:45.

Exercise. Write a function `hhmmss()` such that `hhmmss(secs)` returns a 6-digit integer representing time in the hh:mm:ss 24-hour format where secs is the number of seconds since midnight.

Exercise. Write a function `printhhmmss()` such that `printhhmmss(hhmmss)` prints `hhmmss`, which represents time in the hh:mm:ss 24-hour format, with the hh, mm, ss separated by ':'. For instance `printhhmmss(93045)` prints `09:30:45`.

Exercise. Write a program that accepts two times in the hhmmss 24-hour format and display their difference in the hhmmss 24-hour format. Here's a test run:

```
091410
092500
00:10:50
```

Your program should use the previous functions `seconds_from_midnight()`, `hhmmss()` and `printhhmmss()`.

`main()`

A C/C++ program can have many functions.

In fact you have been using functions since day one: `main()` is a function too.

`main()` is a very special function. When you run your program (in MS Studio .NET, you do that with Ctrl-F5), `main()` is the first function to execute. Therefore every C/C++ program must have a `main()` function.

Of course the curious thing is that `main()` has a return value. Who (or what!!!) does `main()` return the value to?

Without going into details, not only can you get functions to call functions, in fact you can get programs to run programs. The return value of `main()` is sent back to the program that runs it. In the context of a program X running another program Y, the return value reports an error code of the execution of program Y back to program X. The convention is the if program Y executed successfully, the error code returned to X is 0.

Why functions?

There are several reasons:

- **Reduce code duplication by re-using code.**
- **Improve program readability**
- **Improves maintainability**

The last two reasons are similar to the use of constants.

This will be obvious with an example.

```
// this program helps the instructor tracks the
// minimum and maximum number of eyes and heads per
// student in a class of 3

#include <iostream>

int main()
{
    int a, b, c, min, max;

    std::cout << "no. of eyes for each students:";
    std::cin >> a >> b >> c;

    min = a;
    if (min > b) min = b;
    if (min > c) min = c;

    max = a;
    if (max < b) max = b;
    if (max < c) max = c;

    std::cout << "min,max for eyes:"
              << min << ' ' << max << std::endl;

    std::cout << "no. of heads for each students:";
    std::cin >> a >> b >> c;

    min = a;
    if (min > b) min = b;
    if (min > c) min = c;

    max = a;
    if (max < b) max = b;
    if (max < c) max = c;

    std::cout << "min,max for heads:"
              << min << ' ' << max << std::endl;

    return 0;
}
```

You see the computation of `min` twice so ...

```
// this program helps the instructor track the
// minimum and maximum number of eyes and heads per
// student in a class of 3

#include <iostream>


int min(int a, int b, int c)
{
    int m = a;
    if (m > b) m = b;
    if (m > c) m = c;
    return m;
}


int main()
{
    int a, b, c, max;

    std::cout << "no. of eyes for each students:";
    std::cin >> a >> b >> c;

    max = a;
    if (max < b) max = b;
    if (max < c) max = c;

    std::cout << "min,max for eyes:"
              << min(a, b, c) << ' '
              << max << std::endl;

    std::cout << "no. of heads for each students:";
    std::cin >> a >> b >> c;

    max = a;
    if (max < b) max = b;
    if (max < c) max = c;

    std::cout << "min,max for heads:"
              << min(a, b, c) << ' '
              << max << std::endl;

    return 0;
}
```

Exercise. Now's your turn. There's one obvious chunk of code that can be converted to a function. Do it. [Hint: Of course it's the code for the computation of max. Duh.]

Here's a common gotcha :

```
int min(int a, int b, int c)
{
    int m = a;
    if (m > b) m = b;
    if (m > c) m = c;
    return m;
}


int main()
{
    int a, b, c, min, max;

    std::cout << "no. of eyes for each students:";
    std::cin >> a >> b >> c;

    min = min(a, b, c);

    max = a;
    if (max < b) max = b;
    if (max < c) max = c;

    std::cout << "min,max for eyes:"
              << min << ' '
              << max << std::endl;
    ...
}
```

Do you see the problem? The problem is when you're inside `main()`, the `min` is an `int` variable. But you're trying to use `min` as a function as well (see the `min = min(a, b, c)` statement). C++ will be confused because in the scope, the closest `min` is an `int` variable and not a function.

In fact some functions are so commonly used that they are already written for you. You only need to use them (you have to use #include to make them available). You have already used `pow()`, `sqrt()`, `std::setprecision()`, etc. And if you wantt o develop games, then you use a game development library that contains functions to load images from files, blit the image onto a window, load a sound file, play the sound, paint texts, etc.

Testing

Not only is it easier to reuse your code when you broke up your program into re-usable functions, it also makes it easier for you to test your program by testing each separate function.

For instance here's the function isprime from an earlier section:

```
bool isprime(int n)
{
    if (n > 1)
    {
        for (int d = 2; d < n; ++d)
        {
            if (n % d == 0)
            {
                return false
            }
        }
        return true;
    }
    else
    {
        return false
    }
}
```

Primes are used a lot in cryptography and computer security. So perhaps this is part of a very large security system. When building a large system, it's so much easier if you can test the various parts of the system separately. For the case of the above isprime() function, you can write a function to test it:

```
#include <iostream>

bool isprime(int n)
{
    ...
}

// Test isprime(int n) for n = 0, 1, 2, ..., 9.
void test_isprime()
{
    std::cout << (!isprime(0) ? "pass\n" : "FAIL\n")
              << (!isprime(1) ? "pass\n" : "FAIL\n")
              << ( isprime(2) ? "pass\n" : "FAIL\n")
              << ( isprime(3) ? "pass\n" : "FAIL\n")
              << (!isprime(4) ? "pass\n" : "FAIL\n")
              << ( isprime(5) ? "pass\n" : "FAIL\n")
              << (!isprime(6) ? "pass\n" : "FAIL\n")
              << ( isprime(7) ? "pass\n" : "FAIL\n")
              << (!isprime(8) ? "pass\n" : "FAIL\n")
              << (!isprime(9) ? "pass\n" : "FAIL\n")
              << std::endl;
    return;
}

int main()
{
    test_isprime();
    return 0;
}
```

Refactoring

Exercise. Analyze the following code:

```
#include <iostream>

int main()
{
    int a, b, c, d, e, f, g, h;
    std::cin >> a >> b >> c >> d;

    e = a + b + c * d;
    f = a * e + d - b;
    g = e / 5 * f + a;
    h = a * (g - f);

    std::cout << h << std::endl;
}
```

Note that the program prompts the user for 4 int values and displays an int value. Create a function `func()` that reduces the complexity of `main()` in the following way:

- `func()` has the smallest number of parameters to compute the value to be displayed
- The input of the values required to perform the computation in `func()` stays in `main()` - in other words `func()`'s only purpose is to compute the value to display.
- `main()` has the least number of variables so that the program has not changed.

Remember that the new program has only changed in terms of organization of code, but not in its function. Here's the code that actually performs computations:

```
...
    e = a + b + c * d;
    f = a * e + d - b;
    g = e / 5 * f + a;
    h = a * (g - f);
...
```

Here's a skeleton for the new program:

```
#include <iostream>

_____ func(____________________________________)
{
    ...
}


int main()
{
    int __________________________________;
    std::cin >> a >> b >> c >> d;

    std::cout << func(_____________________)
              << std::endl;
}
```

The above exercise where you modify the structure of a code without changing its functionality is frequently called "**refactoring** the code". This is frequently performed on large-scale software to improve the structure of the software. The goal is not to add more features -- it's to simplify the software. Refactoring software is part of software maintenance activity.

In general software undergoes lots of changes in its lifecycle. Sometimes the changes are ad hoc. Without constant refactoring, software "decay" sets in, making the software hard to read, debug, and maintain. In fact the cost of maintenance can be 60-100 times the cost of initial software development. Take CISS465 (Software Engineering) and CISS438 (Object-oriented analysis and design) for more information.

Exercise. The following program

```
#include <iostream>

int main()
{
    int n = 0;
    std::cin >> n;
    for (int i = n-1; i >= 0; i--)
    {
        // Print i spaces
        for (int j = 1; j <= i; j++)
        {
            std::cout << ' ';
        }
        // Print n stars
        for (int j = 1; j <= n; j++)
        {
            std::cout << '*';
        }
        std::cout << std::endl;
    }

    return 0;
}
```

Global scope

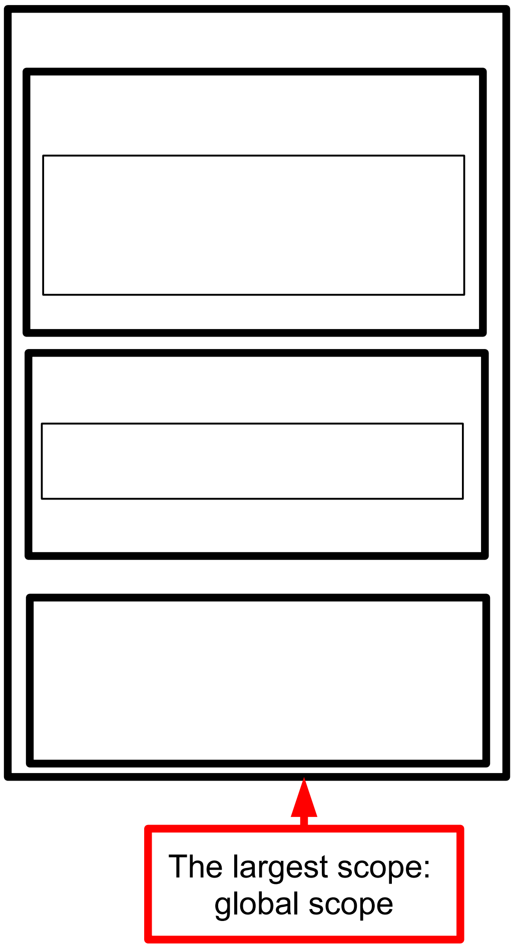
Although the basic scope rules apply to functions (for instance you can't call a function before it's defined) there's still something different between a function name and a variable name.

Up to this point variables are created **within** a function, i.e. variables live inside the memory of functions.

But functions are not. Where does a function live?

Here's a piece of code and a picture showing all the scopes (when they are active):

```
#include <iostream>

int spam()
{
    int x = 42;
    if (...)
    {
        ...
    }
    ...
}

int eggs(double y)
{
    for (...)
        ...
    ...
}

int main()
{
    int x = 24;
    ...
    return 0;
}
```



Note in particular now I'm drawing a scope for the **whole source file**. This is ***also*** a scope – you can think of the whole file as a block if you like. It's called a **global scope**. You can also put names in global scopes. In particular in C/C++ functions have global scopes.

Global variables and global constants

The previous section hints to you that you can of course declare variables in global scopes. Variables declared in the global scope are called **global variables**.

Based on the way C++ search for a variable, can you figure out the output of this program (without running it?):

```
#include <iostream>

int x = 42;


void f()
{
    std::cout << "in f() ..." << x << std::endl;
    ++x;
    std::cout << "in f() ..." << x << std::endl;
    int x = 9;
    std::cout << "in f() ..." << x << std::endl;
    return;
}


int main()
{
    std::cout << "in main() ..." << x << std::endl;
    f();
    std::cout << "in main() ..." << x << std::endl;
    int x = 0;
    std::cout << "in main() ..." << x << std::endl;

    return 0;
}
```

Global variable

Verify yourself!

Now you might then say: “Hey! In that case I don't need to pass values into the function through parameters right? Like this ...”

```
#include <iostream>

double x = 0.0;

void square()
{
    x = x * x;
}

void cube()
{
    x = x * x * x;
}

int main()
{
    double a = 0.0;
    std::cin << a;

    x = a;
    square();
    std::cout << x << std::endl;

    x = a;
    cube();
    std::cout << x << std::endl;
}
```

OK. This is a **really bad practice** because it violates the principle of minimal scope. You have now a `double` variable that is used by everyone!!! (everyone = every function). Side effects will propagate like crazy and debugging an error will be a nightmare (unless if you're masochistic), because if the value of a global variable is corrupted or computed incorrectly and the variable is accessed by hundreds of functions, then you have a much harder problem of tracking down which function is the guilty one.

Using only parameters and return values, you can tell which exactly what goes into the function and what comes out. So if a program has 5000 variable names but a function has three parameters and one return value, you can limit your debugging of that function to four things (at least initially).

So you should almost never use such global variables.

One exception are constants. **Global constants are OK** since being constants they can't be changed anyway, especially **if they are used by many functions**. For instance a business application with the following constants is OK:

```
#include <iostream>

const int COE_EMPLOYEE_CODE = 0;
const int MANAGER_EMPLOYEE_CODE = 1;
const int FULLTIME_EMPLOYEE_CODE = 2;
const int PARTTIME_EMPLOYEE_CODE = 3;

const int PARTTIME_HOURLY_RATE_IN_CENTS = 700;

... code ...
```

Or for a game with a single window this is OK:

```
#include <iostream>

const int W = 640; // width of game window
const int H = 480; // height of game window

... code ...
```

Exercise. Study the following program carefully. Rewrite the program so that global constants are used when appropriate.

```
#include <iostream>


double area(double r)
{
    return 3.1416 * r * r;
}


double circumference(double r)
{
    return 2 * 3.1416 * r;
}


double get_radius()
{
    double r = -1;
    while (r < 0)
    {
      std::cout << "enter radius (>0): ";
      std::cin >> r;
    }
    return r;
}


void area_option()
{
    double r = get_radius();
    std::cout << "area: "
              << area(r) << std::endl;
}


void circumference_option()
{
    double r = get_radius();
    std::cout << "circumference: "
              << circumference(r) << std::endl;
}
```

```
void print_menu()
{
    std::cout << "enter option:\n"
              << "[a] area\n"
              << "[c] circumference\n"
              << "[s] send donation\n"
              << "[q] quit\n";
}


char get_option()
{
    char c = ' ';
    while (c != 'a' && c != 'c'
           && c != 's' && c != 'q')
    {
        std::cout << "option (a, c, s, q): ";
        std::cin >> c;
    }
    return c;
}

int main()
{
    std::cout << "circle calculator!!!\n"
              << "this is not freeware ...\n"
              << "please send your donations to:\n"
              << "yliow@ccis.edu\n"
              << "warning: pi is approximated with "
              << 3.1416 << "\n\n";

    while (1)
    {
        print_menu();
        char c = get_option();
        if (c == 'q' || c == 'Q') break;

        switch (c)
        {
        case 'A':
        case 'a':
            area_option();
            break;

        case 'C':
        case 'c':
            circumference_option();
            break;

        case 's':
        case 'S':
            std::cout << "$1000 is charged to your\n"
                      << "visa card toward\n"
                      << "Dubious Software Co.\n";
            break;
        }
        std::cout << std::endl;
    }

    return 0;
}
```

“Why do I sometimes not see `return` in a function?”

If a function's return type is `void` (i.e. no return value), then once the program has executed the last statement in the body of the function, an automatic return is execute. In other words

```
void printOminousPreamble()
{
    std::cout << "On a dark and stormy night ...";
}
```

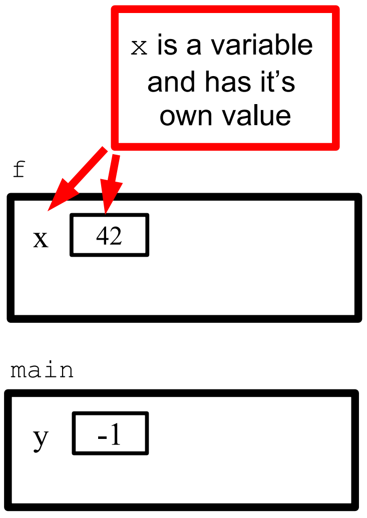
is the same as

```
void printOminousPreamble()
{
    std::cout << "On a dark and stormy night ...";
    return;
}
```

Pass-by-reference

Recall that the parameter `x` in function `f()` is pass-by-value:

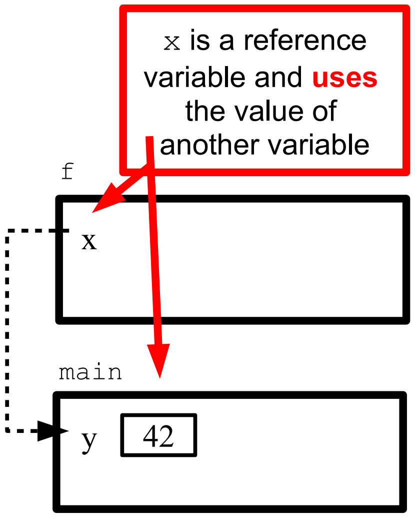
```
void f(int x) // x is a variable and has
              // it's own value. It cannot
              // change y in main().
{
    x = 42;
}


int main()
{
    int y = -1;
    f(y);     // value of y passed to x
    // y is not 42. y is still -1.
    std::cout << y << std::endl;

    return 0;
}
```



Note that the `y` in `main()` is **NOT** changed when you return back to `main()`.

Now try this:

```
void f(int & x) // x is not a variable -- x
                // is a reference or a
                // reference variable. x
                // does not have a value –
                // x uses the value of y.
                // x can change the y in
                // main().
{
    x = 42;
}


int main()
{
    int y = -1;
    f(y); // y is passed to x in f().

    // y is not -1. y is now 42.
    std::cout << y << std::endl;

    return 0;
}
```


x is a reference variable and uses the value of another variable
f
x
main
y
42

In the above example, the variables `x` in `f()` is called a **reference variable**, in particular `x` is an integer reference variable. `x` is different from the regular plain-jane variables: `x` **does not have**

its own memory for keeping a value. Instead, `x` **refers** to other variables. In the above example, `x` refers to the value of `y` in `main()`. A reference variable is an **alias** of another variable – in the above example the `x` in `f()` is just another name for the `y` in `main()`.

When `main()` calls `f()`, the integer reference variable `x` in `f()` will **refer** to the memory of `y`; it **does not receive a value** from `y`. As a matter of fact, referring to the diagram, you see that `x` does not even have its own memory!

We call this type of parameter passing **pass-by-reference**. We say that the `x` in `f()` is a pass-by-reference parameter.

So

- If `y` in `main()` is pass to a variable `x` of `f()`, changing `x` in `f()` does not change the `y` in `main()`.
- If `y` in `main()` is pass to a reference `x` of `f()`, changing `x` in `f()` changes the `y` in `main()`.

Also

- If you use an integer variable as a function parameter, the communication into that function uses pass-by-value.
- If you use a reference as a function parameter, the communication into that function uses pass-by-reference.

Here's another example. First run this program:

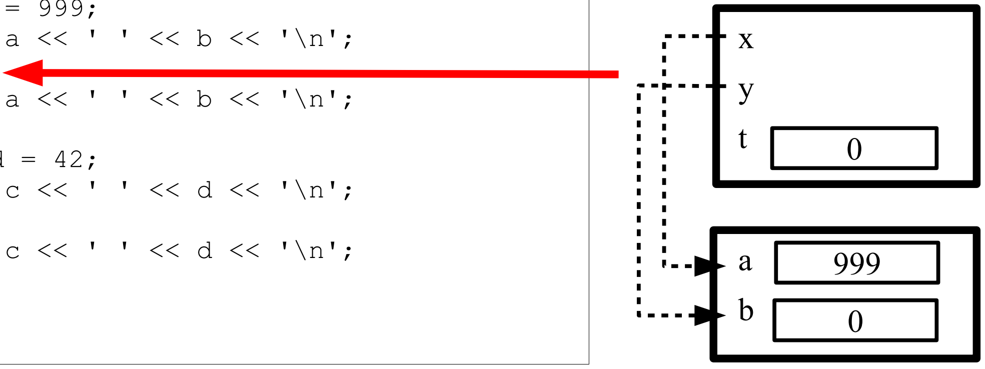
```
#include <iostream>

void swap(int x, int y)
{
    int t = x;
    x = y;
    y =t;
}


int main()
{
    int a = 0, b = 999;
    std::cout << a << ' ' << b << '\n';
    swap(a, b)
    std::cout << a << ' ' << b << '\n';

    int c = -1, d = 42;
    std::cout << c << ' ' << d << '\n';
    swap(c, d)
    std::cout << c << ' ' << d << '\n';

    return 0;
}
```

Run it, stare at the output, and study the program. Both parameters `x` and `y` in `swap()` uses pass-by-value.

Now compare it with this:

```
#include <iostream>

void swap(int & x, int & y)
{
    int t = x;
    x = y;
    y =t;
}

int main()
{
    int a = 0, b = 999;
    std::cout << a << ' ' << b << '\n';
    swap(a, b)
    std::cout << a << ' ' << b << '\n';

    int c = -1, d = 42;
    std::cout << c << ' ' << d << '\n';
    swap(c, d)
    std::cout << c << ' ' << d << '\n';

    return 0;
}
```

Memory model just before return



Get it?

Exercise. What is the output?

```
#include <iostream>

void increment(int x)
{
    ++x;
}

int main()
{
    int x = 0;
    increment(x);
    std::cout << x << '\n';
    return 0;
}
```

Exercise. Fix this program so that it works:

```
#include <iostream>
```

```
void increment(int & x)
{
    ++x;
}


int main()
{
    int a = 42;
    increment(a);
    std::cout << a << '\n'; // should see 43
    return 0;
}
```

(Of course this is just for demo. In a real world program, you should use ++ instead of writing a function for it!)

Exercise. The following does not work – see the comment below. Fix it.

```
#include <iostream>

void clear(int x)
{
    x = 0;
}


int main()
{
    int x = 42;
    clear(x);
    std::cout << x << '\n'; // Should be 0.
    return 0;
}
```

Exercise. The following does not work – see the comment below. Fix it.

```
#include <iostream>

void double_it(double x)
{
    x *= 2;
}

int main()
{
    double x = 1.1;
    double_it(x);
    std::cout << x << '\n'; // Should be 2.2.
    x = 5;
    double_it(x);
    std::cout << x << '\n'; // Should be 10.
    return 0;
}
```

Exercise. Complete this program:

```
// sort x, y, z in ascending order using bubblesort
void sort(int & x, int & y, int & z)
{
  ... CODE ...
}

int main()
{
  int x = 3, y = 5, z = 2;
  sort(x, y, z);
  std::cout << x << ' ' << y << z << ' '
            << std::endl; // should get: 2 3 5
  return 0;
}
```

Important advice: In general we use references only when a function is meant to change the value of a variable that is passed in. We do not want accidental changes in the function to propagate back to the caller. In other words, use references only when you need to. Therefore you should minimize the use of reference variables.

Exercise. One of the parameters in `f()` need not be a reference. Which one? You have 2 seconds.

```
int f(int & a, double & b)
{
    b += a;
    return a * b
}
```

Exercise. Is the person who wrote this function a good programmer? Why? You have 2 seconds.

```
double average(int x, int y)
{
    return (x + y) / 2.0;
}
```

Return value or pass-by-reference???

Now we have a dilemma …

At this point, there are two ways of changing the value of a variable: The following is our simple `square()` function:

```
#include <iostream>

double square(double x)
{
    return x * x;
}


int main()
{
    double the_square;
    the_square = square(3.14);
    std::cout << the_square << std::endl;
    return 0;
}
```

Now, after studying references, you know that you can also do this:

```
void square(double & the_square, double x)
{
    the_square = x * x;
}


int main()
{
    double the_square;
    square(the_square, 3.14);
    std::cout << the_square << std::endl;
    return 0;
}
```

Make sure you run it.

Which should you use??? (Don't you hate making decisions ...)

In general, if your function is like a mathematical function that computes a value, then you should use a return value. (Principle of least surprise.)

For instance look at the above example. Notice how clumsy is the usage of the `square()` function. For instance if you just want the value of the square of 3.14, you can do this:

```
#include <iostream>

double square(double x)
{
    return x * x;
}
```

```
int main()
{
    std::cout << square(3,14) << std::endl;
    return 0;
}
```

You don't have to declare a variable to store the square of 3.14. The method of references requires a variable – you have no choice.

If you have a function that computes two values, then you might want to have to have two pass-by-reference variables. Of course you can have two different functions. Here's an example:

```
void min_max(int x, int y, int & min, int & max)
{
    min = (x <= y ? x : y);
    max = (x >= y ? x : y);
}
```

This function computes the max and min of two inputs. Of course you **cannot** return two values in a function:

```
int, int min_max(int x, int y)
{
    min = (x <= y ? x : y);
    max = (x >= y ? x : y);
    return min, max;
}
```

WRONG!!!

So in this case you have to a function that returns two values in the sense of modifying two variables which are passed into the function by reference.

However this means that one fine day, if you really need to compute only the minimum (and not maximum), you will run into some ugly coding. This is like going to a car dealer that insist that whenever you buy a car from them you must buy your car insurance policy through them as well.

A basic principle of good software design is to design each function to perform one task and not two. There are however cases where the values computed cannot be viewed as separate entities – they are viewed as a package of values that has to be used together at all times.

Of course you can have two different functions:

```
int min(int x, int y)
{
    return (x <= y ? x : y);
}

int max(int x, int y)
{
    return (x >= y ? x : y);
}
```

So if you have a function that computes and returns more than one value, you have **two choices**:

- "Return" the values through **multiple reference variables**
- Redesign it into **multiple functions**, each computing and returning one value

Now if you look at our `swap()` function

```
void swap(int & a, int & b)
{
    int t = a;
    a = b;
    b = t;
}
```

It does not perform a numerical computation; there is really no return value. It performs a transformation on variables. Note that the values of both variables should be change. There's no reason to change one of the variables and not the other. So in this case, it makes sense to have one function and not two, even though technically two things happened. That's why the design of this function is reasonable.

There are other uses of reference variables. We'll come back to this concept again later in a set of notes just on references.

Style

Instead of

```
int & x
```

another more common style is

```
int &x;
```

Function prototype

First, try this:

```
#include <stream>

void mamamia()
{
    std::cout << "ma ma mia\n";
}

int main()
{
    mamamia();
    return 0;
}
```

Next, try this:

```
#include <iostream>

int main()
{
    mamamia();
    return 0;
}

void mamamia()
{
    std::cout << "ma ma mia\n";
}
```

Of course the program won't compile. The reason is because the function `mamamia()` is defined after the function call of `mamamia()` in `main()`. This is not new. I have already talked about it.

This means that we need to arrange our function code so that the definition of a function is before the function is called. There's a way around this issue ...

Try this:

```
#include <iostream>

void mamamia();

int main()
{
    mamamia();
    return 0;
}

void mamamia()
{
    std::cout << "ma ma mia\n";
}
```

Voila! It works.

The statement in the above code:

```
...

void mamamia();

...
```

is called a **function prototype**.

A **function prototype** is just the header of a function (without the function body) which ends with a semicolon – it's a complete statement. The function prototype announces the existence of a function somewhere in the code. That way, you can define the function anywhere in your source file. This will save you some time on re-organizing your code. That's it.

Here's another example. This will not compile:

```
#include <iostream>

int main()
{
    int a = 0, b = 999;
    swap(a, b)
    std::cout << a << ' ' << b << '\n';
    return 0;
}


void swap(int & x, int & y)
{
    int t = x;
    x = y;
    y =t;
}
```

I can add a function prototype for the `swap()` function:

```
#include <iostream>

void swap(int & x, int & y);

int main()
{
    int a = 0, b = 999;
    swap(a, b)
    std::cout << a << ' ' << b << '\n';
    return 0;
}

```

```
void swap(int & x, int & y)
{
    int t = x;
    x = y;
    y =t;
}
```

Now it compiles.

Another thing to note is that in function prototypes, **parameter names** are actually **optional**. Try this:

```
#include <iostream>

void swap(int &, int &);

int main()
{
    int a = 0, b = 999;
    swap(a, b)
    std::cout << a << ' ' << b << '\n';
    return 0;
}


void swap(int & x, int & y)
{
    int t = x;
    x = y;
    y =t;
}
```

Again it works.

As I mentioned earlier, using function prototypes relieves you of the pain of having to spend time organizing your code so that functions definitions can be anywhere in your source file. There's actually more to it. There's a later set of notes on function prototype. But this is enough for now. We'll come back function prototypes again later.

How do you write function prototypes for pass-by-reference parameters? Easy! It's the same as your other variables. For instance if you have this function:

```
void f(int a, double & b, char & c, const bool & b)
{
    ... code ...
}
```

The function prototype is just

```
void f(int, double &, char &, const bool &);
```

That's all there is to it. So there's nothing new here.

Exercise. Given this code

```
#include <iostream>

int f(int & a, double & b)
{
    a++;
    b += a;
    return a * b
}

int main()
{
    int x = 42;
    double y = 1;
    int z = f(x, y);
    std::cout << x << ' ' << y << ' ' << z
              << std::endl;
    return 0;
}
```

Rewrite it into this form:

```
#include <iostream>

// prototype of f() here

int main()
{
    int x = 42;
    double y = 1;
    int z = f(x, y);
    std::cout << x << ' ' << y << ' ' << z
              << std::endl;
    return 0;
}

int f(int & a, double & b)
{
    a++;
    b += a;
    return a * b
}
```

Run your program to make sure that you did it correctly.

Exercise. Complete the following

```
// function prototype for rotate_right

int main()
{
    int x = 42;
    int y = 1;
    int z = -3;
    rotate_right(x, y, z); // x,y,z becomes -3,42,1
    std::cout << x << ' ' << y << << ' ' << z
              << std::endl;
    return 0;
}

// function definition for rotate_right
```

Test your program.

Exercise. Complete the following

```
// function prototype for rotate_left

int main()
{
    int x = 42;
    int y = 1;
    int z = -3;
    rotate_left(x, y, z); // x,y,z become 1,-3,42
    std::cout << x << ' ' << y << ' ' << z << '\n';
    return 0;
}

// function definition for rotate_right
```

Test your program.

Exercise. Complete the following

```
// function prototype for digits

int main()
{
    int x = 152;
    int digit2, digit1, digit0;
    digits(x, digit2, digit1, digit0);
    // digit2,digit1,digit0 become 1,5,2
    std::cout << digit2 << ' ' << digit1 << ' '
              << digit0 << '\n';
    return 0;
}

// function definition for digits
```

Test your program.

Summary

A function looks like this:

[return type] [func name]([param type] [var name])
{
 [body of function]
}

There can be more than one parameter type and name. A parameter is just a variable that receives a value passed into the function. The return type can be any valid type including `void` if the function does not return a value.

The function body can be any valid C++ statement. To return a value the function executes

```
return [expression];
```

in its body. If the function does not return a value, executing

```
return;
```

will return to the point of function call.

If a function has return type void, return need not be present. In that case when the execution reached the end of the function block, a return is executed.

Functions cannot be nested.

The largest scope in a program file is the global scope. You can declare variables and constants in the global scope. These are called global variables and global constants respectively. However you should avoid (if possible) the declaration of global variables.

On returning from a function, since the program leaves the scope of the function, all variables created within the function's scope are destroyed.

A parameter is pass-by-value if the parameter has its own value. A parameter is pass-by-reference if the parameter does not have its own value but references the value of a variable that is passed to that parameter. Changing the value of the parameter that is pass-by-reference will change the value of the variable passed to the parameter. By default, parameters of basic types (int, double, char, bool) are pass-by-value. The following parameter `x` is pass-by-value:

```
void f(..., int x, ...) { ... }
```

while the following parameter `y` is pass-by-reference:

```
void g(..., int & x, ...) { ... }
```

A function prototype is just the header of a function as a statement. A function prototype is usually placed near the top of source files in order to indicate to the compiler the existence that function. If a function prototype is seen by the compiler, then calling that function will be compiled correctly by the compiler even if the function definition is below the point where the function call is made.

Exercise

Q1. Write a function `divides()` so that `divides(d, n)` return `true` exactly when `d` divides `n`. Test your code.

Q2. Write a function `average()` so that `average(a, b, c)` returns the average of doubles `a`, `b`, and `c`.

Q3. Write a function `num_digits()` so that `num_digits(n)` returns the number of digits of `n`.

Q4. Write a function `digit()` such that `digit(n, i)` returns the `i`-th digit of `n`. For instance `digit(834, 0)` return `4`, `digit(834, 1)` returns `3`, `digit(834, 2)` returns `8`, and `digit(834, 3)` returns `0`.

Q5. Write a function `prettyprintint()` such that `prettyprintint(n)` prints `n` with commas separating groups of 3 digits. For instance `prettyprintint(12345678)` prints `12,345,678` (note the commas).

Q6. Write a function `nextprime()` such `nextprime(n)` returns the smallest prime which is at least `n`. For instance `nextprime(6)` return `7` and `nextprime(11)` returns `11.`

Q7. Write a function `randunit()` such that `randunit()` returns a random double d in the interval [0.0, 1.0].

Q8. Write a function `randrange()` such that `randrange(a, b)` returns a random int in the interval [a, b – 1] where a and b are integers.

Q9 Write a function randdouble() such that randdouble(a, b) where a and b are doubles returns a random double d such that a <= d and d <= b.