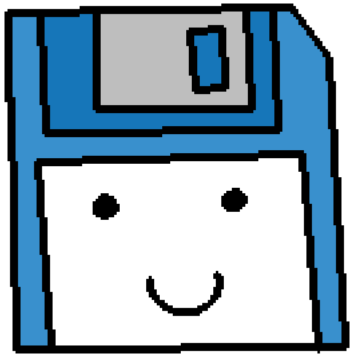
All About Functions

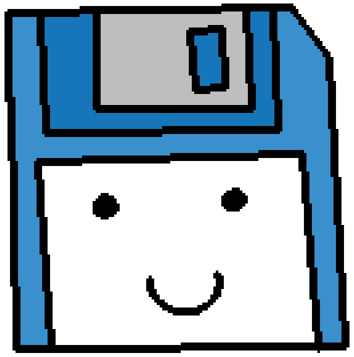
Written by Rachel J. Morris, last updated 2015-01-24

Functions are an important part of programming, so there is a lot to cover this time.



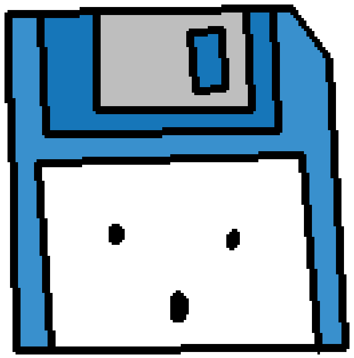
Pay attention to the vocabulary of functions. Understanding the vocabulary is important so that you can clearly discuss functions with other programmers.

Functions allow us to pull functionality into separate areas of the program, so we don't have a big, cluttered mess in **`main()`**.



And because Functions have names, they also allow us to keep our program clean by describing what the code inside a function does.

What is a function?



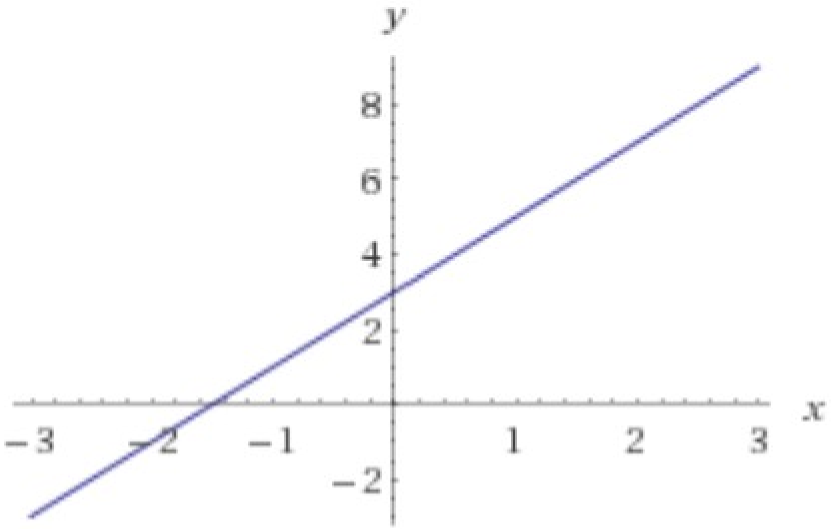
What is a function?

$$f(x) = 2x + 3$$

In math, we use functions a lot.

x	f(x)
0	3
1	5
2	7

x is a variable we plug into the function **f(x)**, and we get some output (which usually is a **y** coordinate)



Graph generated with Wolfram Alpha

What is a function?

$f(x) = 2x + 3$

```
float f( float x ) { return 2x + 3; }
```

Functions in C++ look similar to in math, except that we need to specify data-types for the **return value** of the function and for the **parameter**.

What is a function?

$$f(x) = 2x + 3$$

```
float f( float x ) { return 2x + 3; }
```

Data type of the returned value

Function name

Parameter data type and name

Function body

What is a function?

```
int main()
{
    cout << "Hello, World!" << endl;

    return 0;
}
```

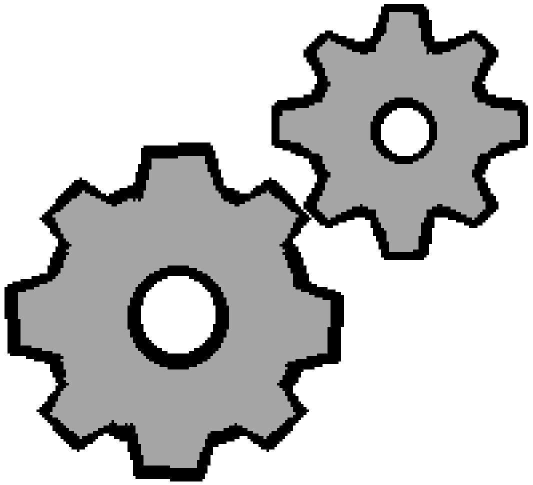
Our `main()` is a function. Notice that we specify its return-type (an integer), and at the end of the program, we `return 0;` close the program.

However, we don't call the **main()** function ourselves.

What is a function?

When we write a **function**, we will specify:

- **Output:** What kind of information will be returned (int, float, bool, nothing, etc.)
- **Input:** What kind of information will be “passed in” to work with.
- The name of the function, which we will use to **call** it from other places in the program.

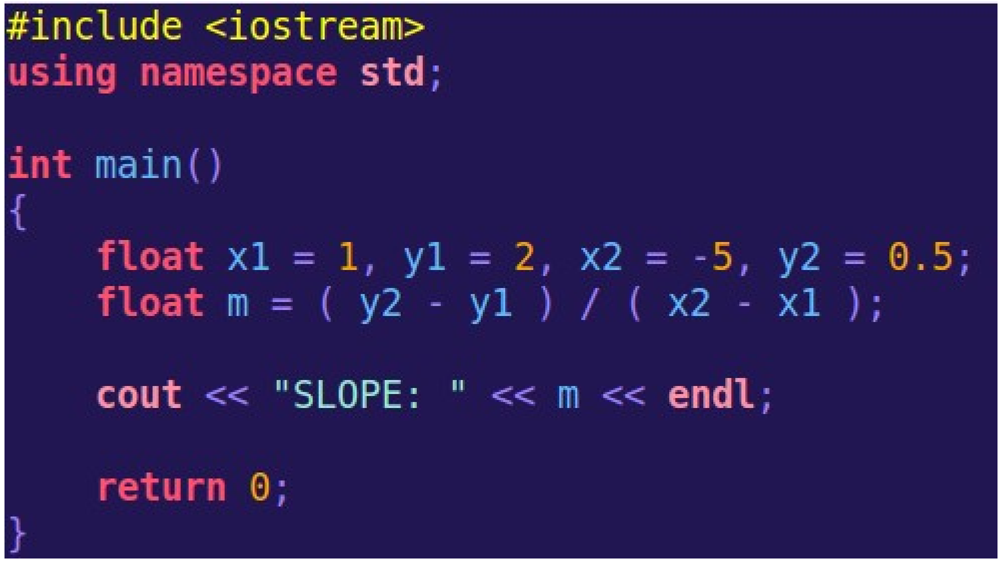


Function name:
Slope

Input:
x1, x2, y1, and y2. (floats)

Output:
The slope between the two points. (float)

What is a function?



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

int main()
{
    float x1 = 1, y1 = 2, x2 = -5, y2 = 0.5;
    float m = ( y2 - y1 ) / ( x2 - x1 );

    cout << "SLOPE: " << m << endl;

    return 0;
}
```

So far, we've been writing all of our code within main.

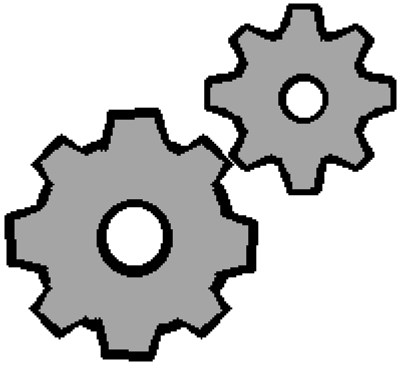
If we want to calculate slope more than once, we have to keep re-typing the formula.

The more you write duplicate code, the higher the chances of causing errors, like...

When making changes to the formula, forgetting to update certain spots.

Accidentally making a typo in one of the areas, resulting in a logic error.

What is a function?



```
float Slope( float x1, float y1, float x2, float y2 )
{
    return ( y2 - y1 ) / ( x2 - x1 );
}
```

This also keeps our code **cleaner**, because now we have a **function name**, that describes what is going on with the code on the inside.

We can write a function to do our calculation for us, and then we simply need to call **Slope** any time we need to calculate it:

```
float slope = Slope( 1, 2, 0.5, 1 );
cout << "Slope: " << slope << endl;

slope = Slope( 5, 4, 0.3, 0.2 );
cout << "Slope: " << slope << endl;

slope = Slope( 0, 0, 100, 0 );
cout << "Slope: " << slope << endl;
```

What is a function?

You may have already used some functions that are part of the C++ standard library, such as:

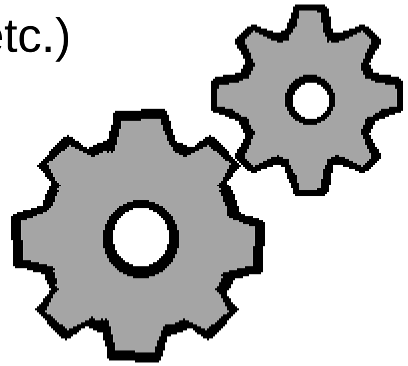
```cpp
int randomNumber = rand() % 100;
```

`rand()` takes no input, but will return an **integer** output.

```cpp
float root = sqrt( 4 );
```

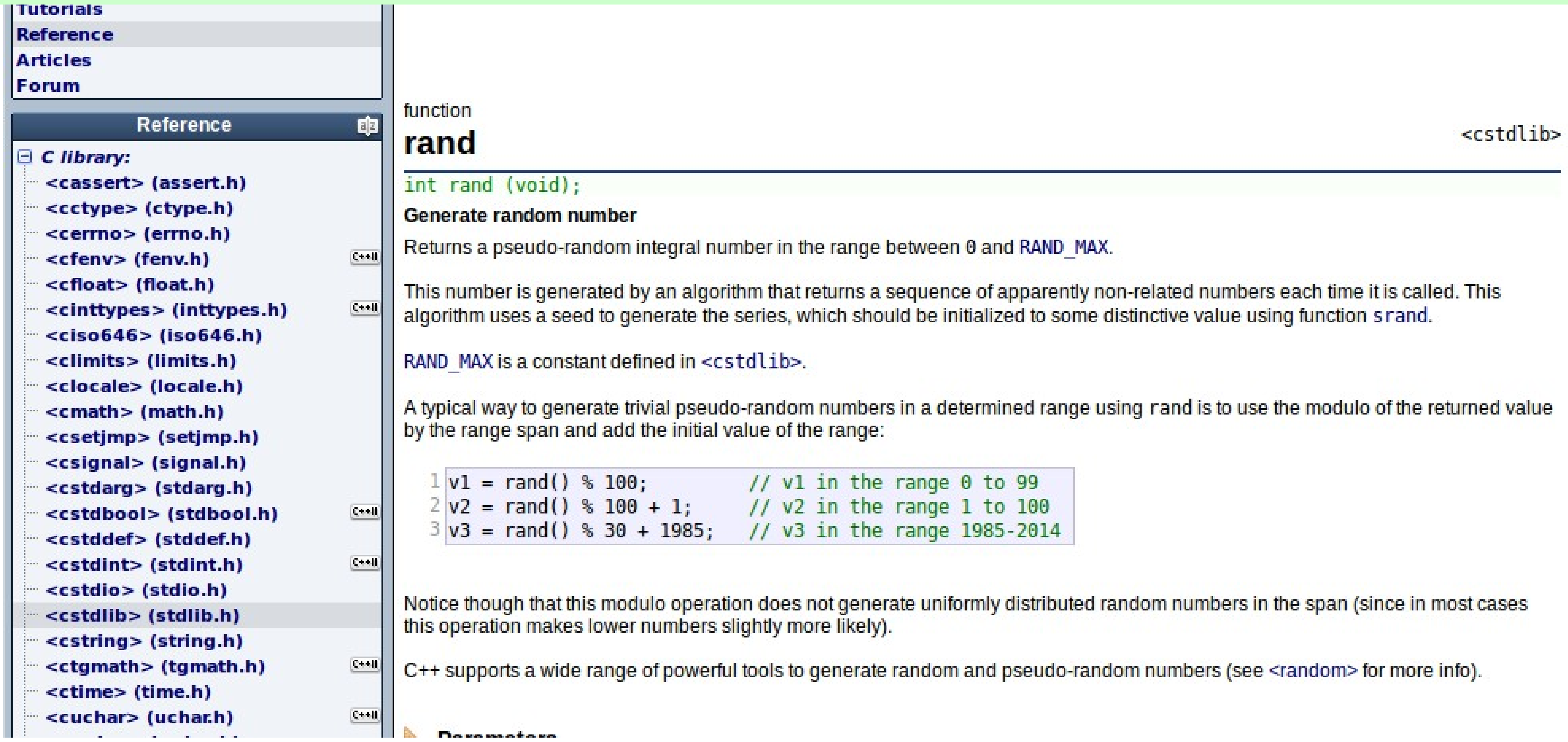
$\sqrt{4} = 2$

`sqrt()` takes an input of type **double** (but you can pass in an int, float, etc.)
And will return a **double**, which is the square root of the value passed in.



What is a function?

You can look up these functions on the **cplusplus.com** reference website.
It will tell you the input, output, and sample code.



function
rand <cstdlib>

```
int rand (void);
```

Generate random number

Returns a pseudo-random integral number in the range between 0 and RAND_MAX.

This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses a seed to generate the series, which should be initialized to some distinctive value using function srand.

RAND_MAX is a constant defined in <cstdlib>.

A typical way to generate trivial pseudo-random numbers in a determined range using rand is to use the modulo of the returned value by the range span and add the initial value of the range:

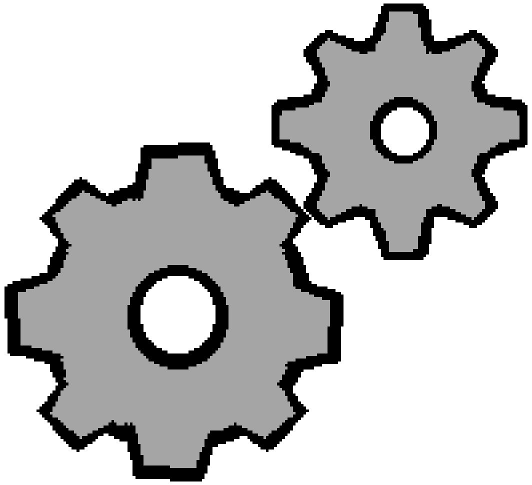
```
v1 = rand() % 100;         // v1 in the range 0 to 99
v2 = rand() % 100 + 1;     // v2 in the range 1 to 100
v3 = rand() % 30 + 1985;   // v3 in the range 1985-2014
```

Notice though that this modulo operation does not generate uniformly distributed random numbers in the span (since in most cases this operation makes lower numbers slightly more likely).

C++ supports a wide range of powerful tools to generate random and pseudo-random numbers (see <random> for more info).

Parameters

Anatomy of a Function



Anatomy of a Function
Function Declaration

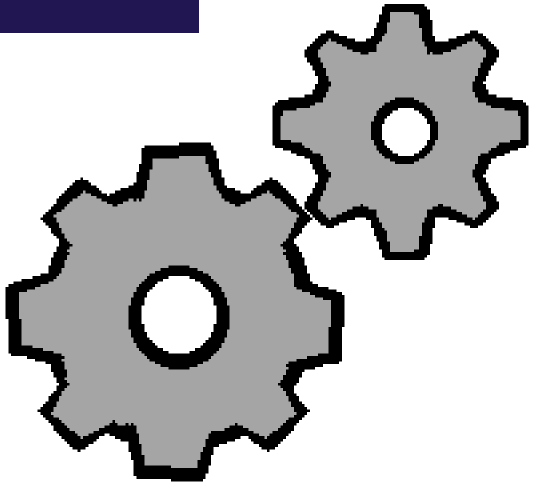
Function Declaration

A function declaration is simply telling C++ that there will be a function, with some **name** and **parameters**, but the function declaration **does not** contain the internal code.

```
float QuadraticX1( float a, float b, float c );
float QuadraticX2( float a, float b, float c );
```

It is similar to a **variable declaration**, because it ends with a semi-colon.

However, it still contains parenthesis and a list of **parameters**.

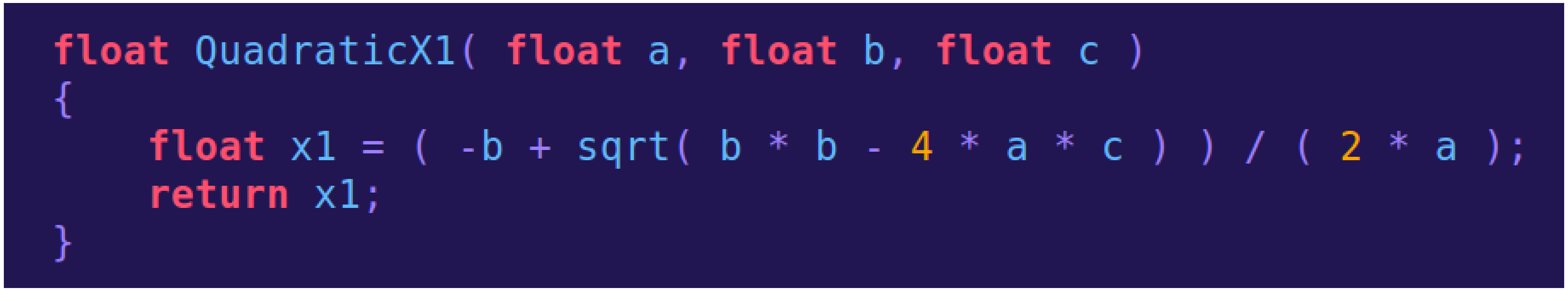


Anatomy of a Function
Function Definition

Function Definition

A function definition is where we actually add code to the inside of our function.

The function body is contained within a pair of curly-braces, { and }, and does not have a semi-colon at the end.



```
float QuadraticX1( float a, float b, float c )
{
    float x1 = ( -b + sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
    return x1;
}
```

Anatomy of a Function
Function Header

Function Declaration

```
float QuadraticX1( float a, float b, float c );
float QuadraticX2( float a, float b, float c );
```

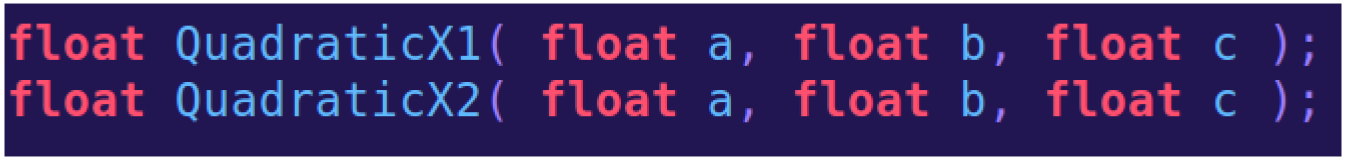
Function Definition

```
float QuadraticX1( float a, float b, float c )
{
    float x1 = ( -b + sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
    return x1;
}
```

The line that specifies the **function return type**, **function name**, and **function parameters**, is called the **function header.**

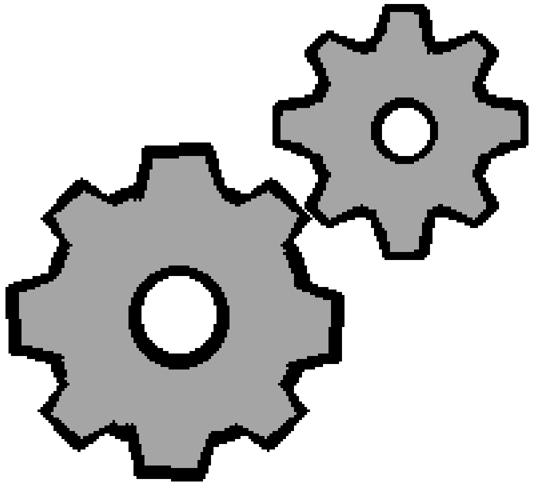
Anatomy of a Function
Function Parameter(s)

The variables listed within the (and) of a function declaration or definition are called **parameters**.



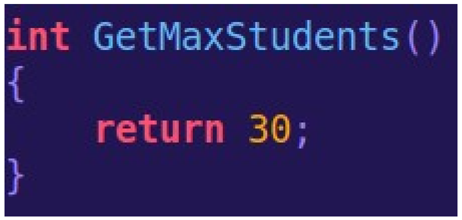
```
float QuadraticX1( float a, float b, float c );
float QuadraticX2( float a, float b, float c );
```

Here, **a**, **b**, and **c** are parameters.

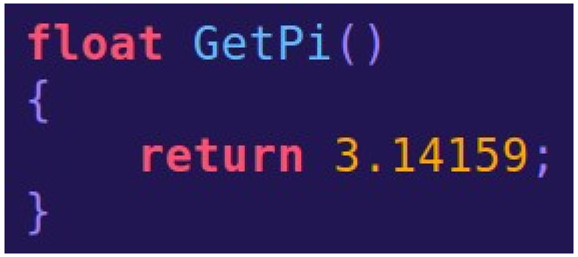


Anatomy of a Function
Function Parameter(s)

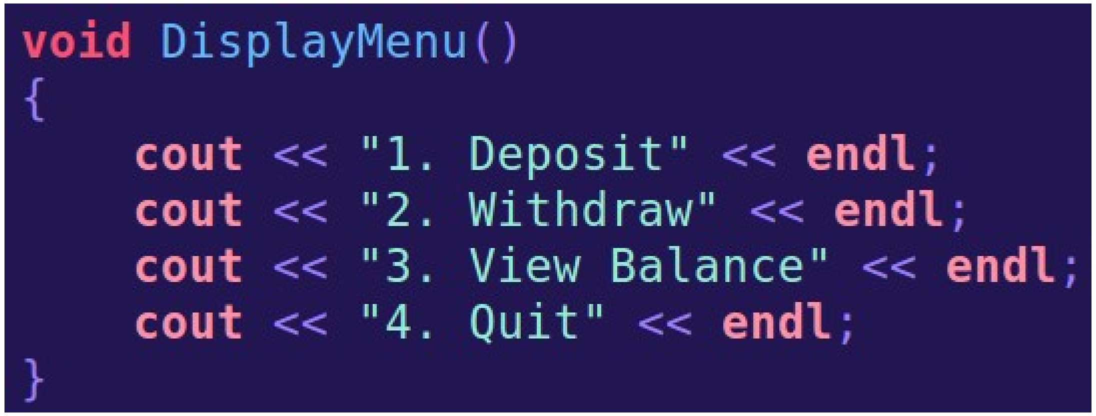
A function isn't required to have parameters.
If the function does not need any input, the parenthesis are empty.



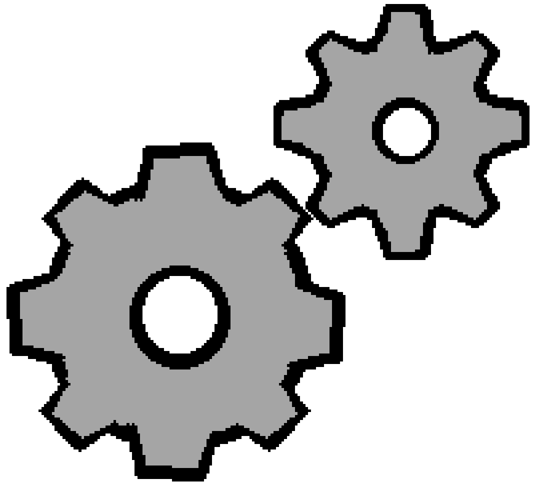
```
int GetMaxStudents()
{
    return 30;
}
```



```
float GetPi()
{
    return 3.14159;
}
```

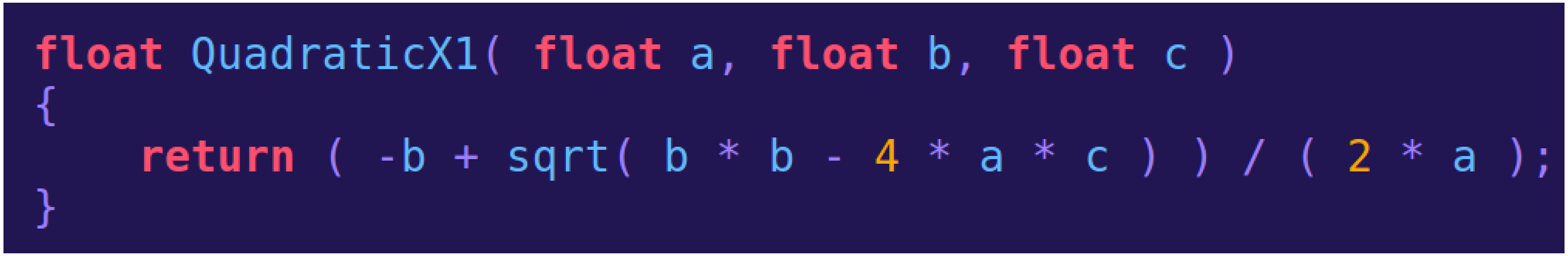


```
void DisplayMenu()
{
    cout << "1. Deposit" << endl;
    cout << "2. Withdraw" << endl;
    cout << "3. View Balance" << endl;
    cout << "4. Quit" << endl;
}
```

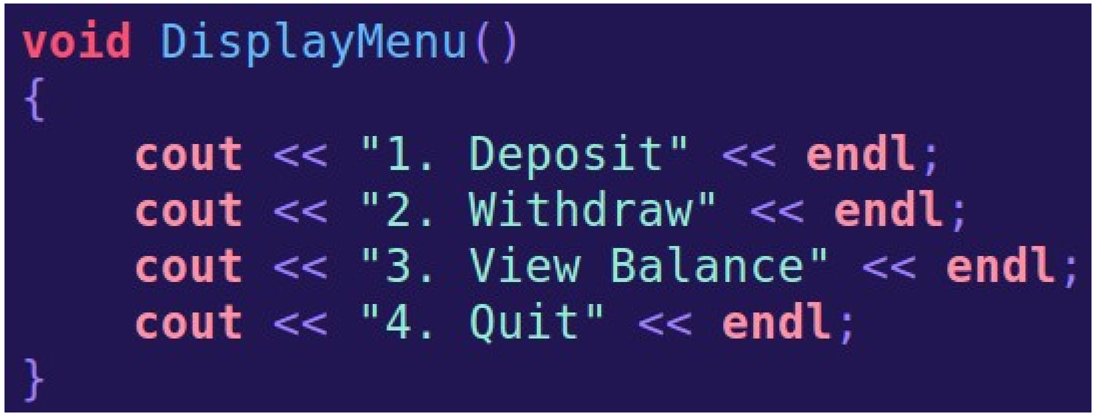


Anatomy of a Function
Function Body

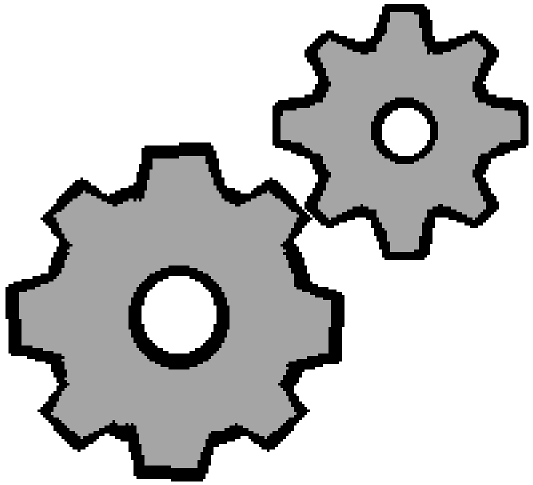
The function body is the code contained within the curly-braces { }.
This is the code that will be executed any time the function is **called**.



```
float QuadraticX1( float a, float b, float c )
{
    return ( -b + sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
}
```

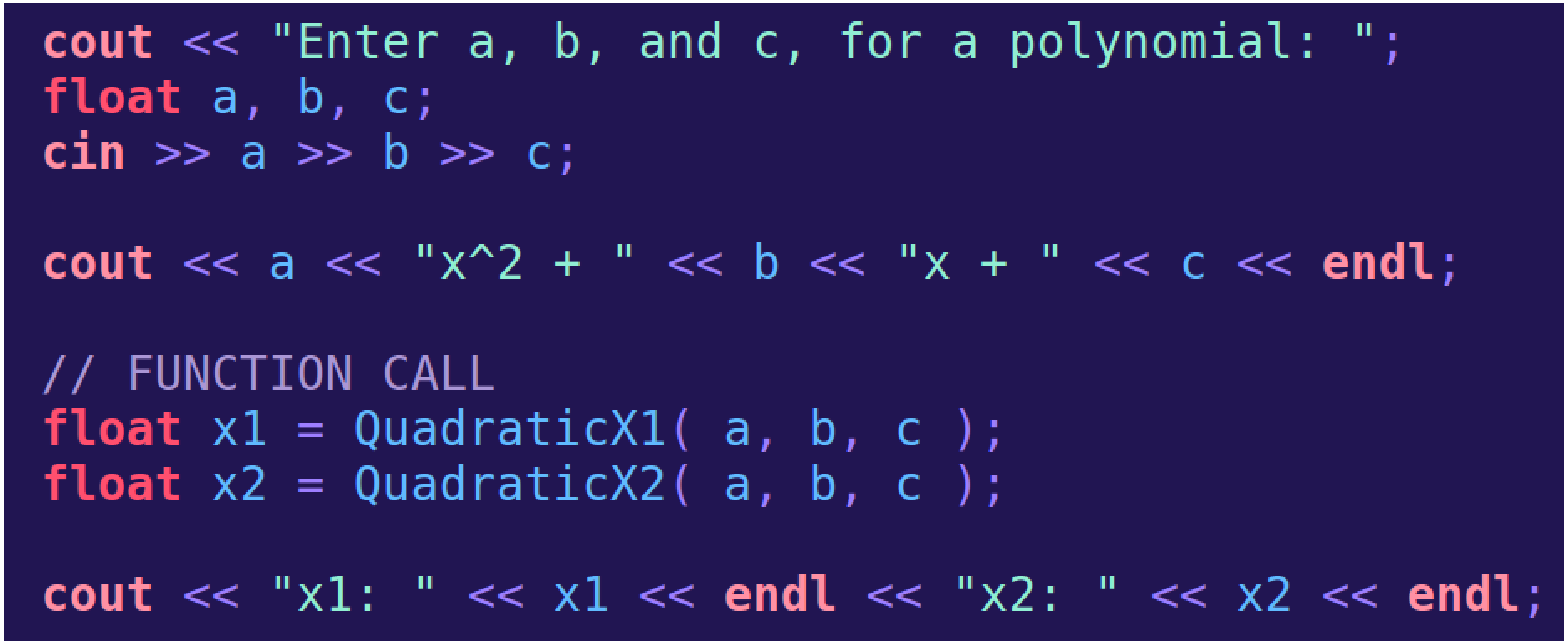


```
void DisplayMenu()
{
    cout << "1. Deposit" << endl;
    cout << "2. Withdraw" << endl;
    cout << "3. View Balance" << endl;
    cout << "4. Quit" << endl;
}
```



Anatomy of a Function
Function Call

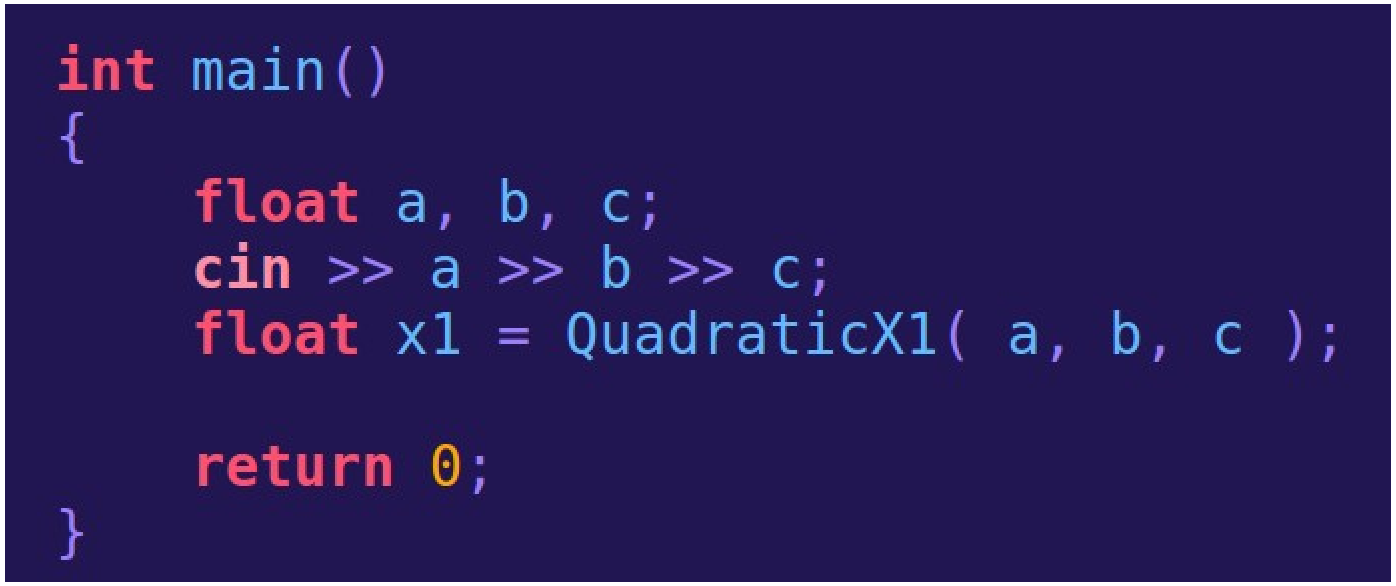
A function is **called** whenever we use its name within the program, and pass **arguments** to that function.



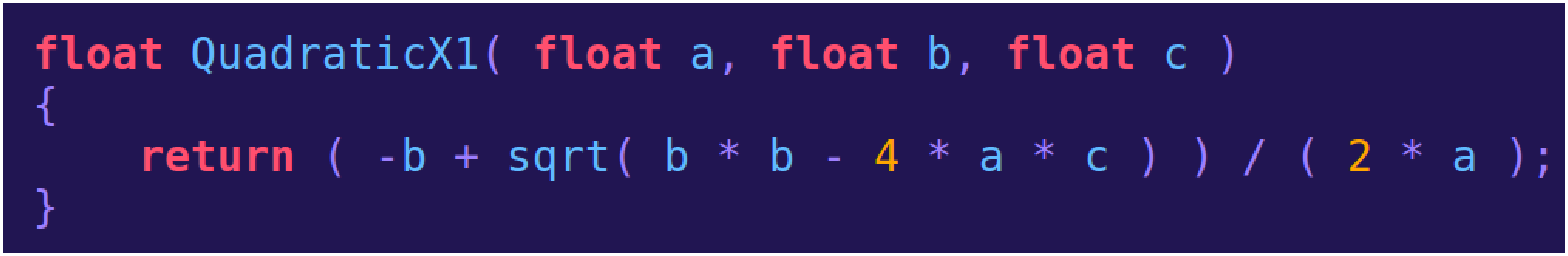
```
cout << "Enter a, b, and c, for a polynomial: ";
float a, b, c;
cin >> a >> b >> c;

cout << a << "x^2 + " << b << "x + " << c << endl;

// FUNCTION CALL
float x1 = QuadraticX1( a, b, c );
float x2 = QuadraticX2( a, b, c );

cout << "x1: " << x1 << endl << "x2: " << x2 << endl;
```

Anatomy of a Function
Function Call

We can store the **return value** of our function in a variable. Here, the function calculates one of the **x** solutions, and we are storing it in the variable **`x1`**.



```
int main()
{
    float a, b, c;
    cin >> a >> b >> c;
    float x1 = QuadraticX1( a, b, c );

    return 0;
}
```



```
float QuadraticX1( float a, float b, float c )
{
    return ( -b + sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
}
```

Anatomy of a Function
Function Call

In our **main()**, we have declared **a, b,** and **c**.
We **pass** the values of a, b, and c to our **QuadraticX1** function as the input,
And we get the output based on the quadratic formula.

```
int main()
{
    float a, b, c;
    cin >> a >> b >> c;
    float x1 = QuadraticX1( a, b, c );

    return 0;
}
```

When we are **calling** a function, the variables we pass into the function are called

arguments

```
float QuadraticX1( float a, float b, float c )
{
    return ( -b + sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
}
```

Anatomy of a Function
Function Call

```
int main()
{
    float a, b, c;
    cin >> a >> b >> c;
    float x1 = QuadraticX1( a, b, c );

    return 0;
}
```

When we're writing our **function definition**, the input values the function is expecting is called

parameters

```
float QuadraticX1( float a, float b, float c )
{
    return ( -b + sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
}
```

Anatomy of a Function
Parameters vs. Arguments

Function Call (Arguments)

```
float x1 = QuadraticX1( a, b, c );
```

Arguments

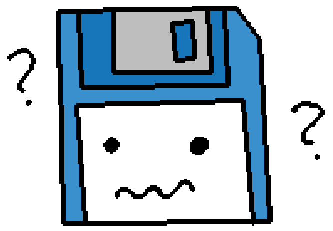
The **variables**, that we are passing as input to the function.

Function Definition or Declaration (Parameters)

```
float QuadraticX1( float a, float b, float c )
```

Parameters

The **input** that the function is expecting.

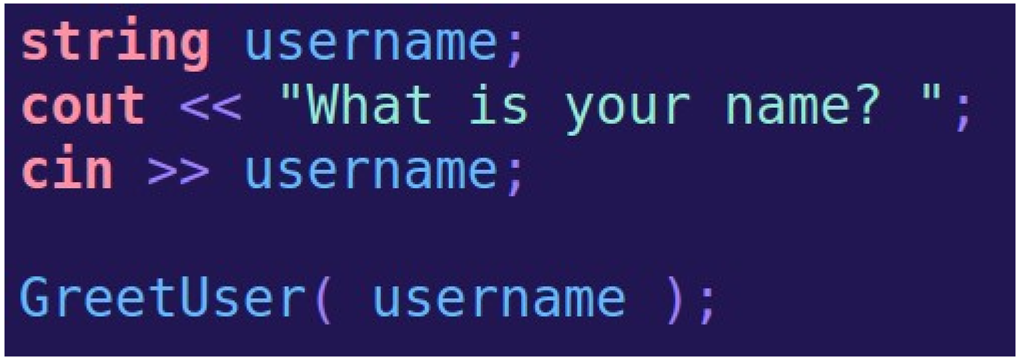


It may seem confusing, but try to remember the difference!

Anatomy of a Function Parameters vs. Arguments

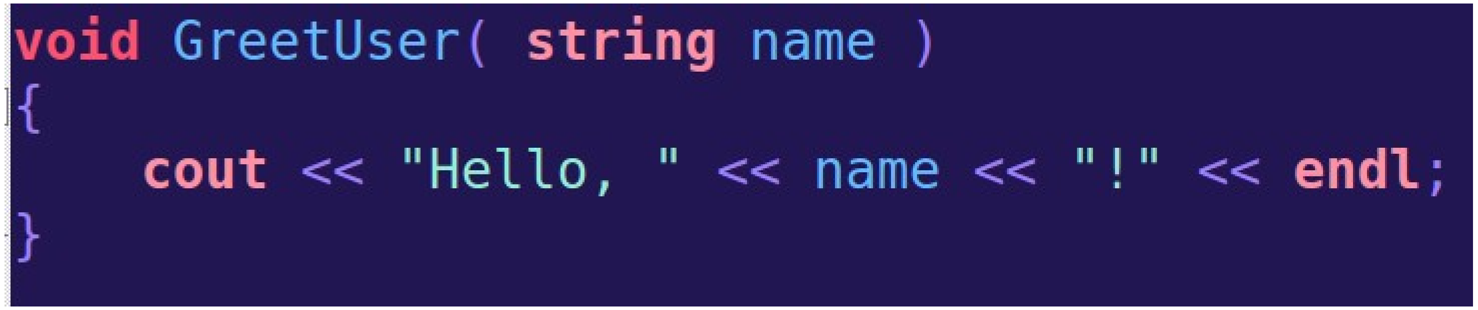
Additionally, the names of the variables passed in as **arguments** do not need to match the names of the **parameters**.

Function Call (Argument)



```
string username;
cout << "What is your name? ";
cin >> username;

GreetUser( username );
```

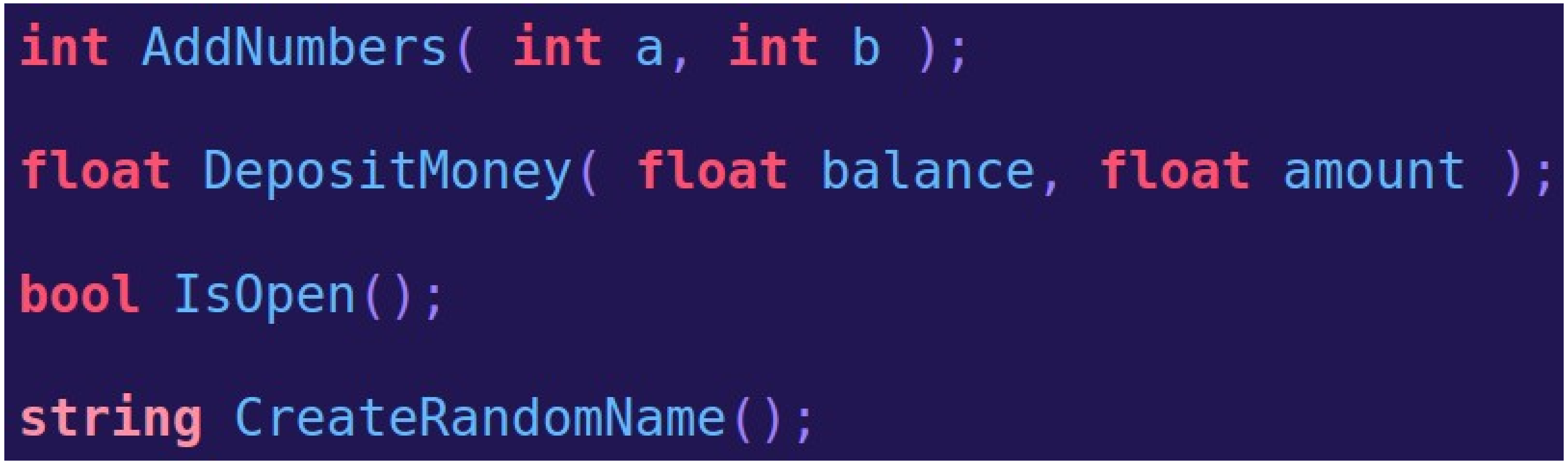
Function Definition or Declaration (Parameter)



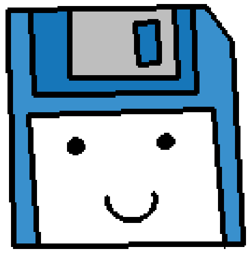
```
void GreetUser( string name )
{
    cout << "Hello, " << name << "!" << endl;
}
```

Anatomy of a Function return types

Any data type we can use for **variables**, we can also use as a **return type** for our functions.

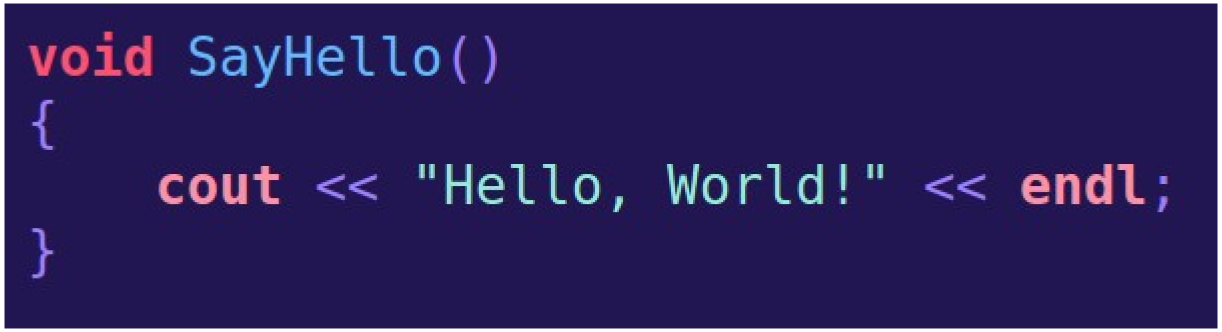


```
int AddNumbers( int a, int b );

float DepositMoney( float balance, float amount );

bool IsOpen();

string CreateRandomName();
```

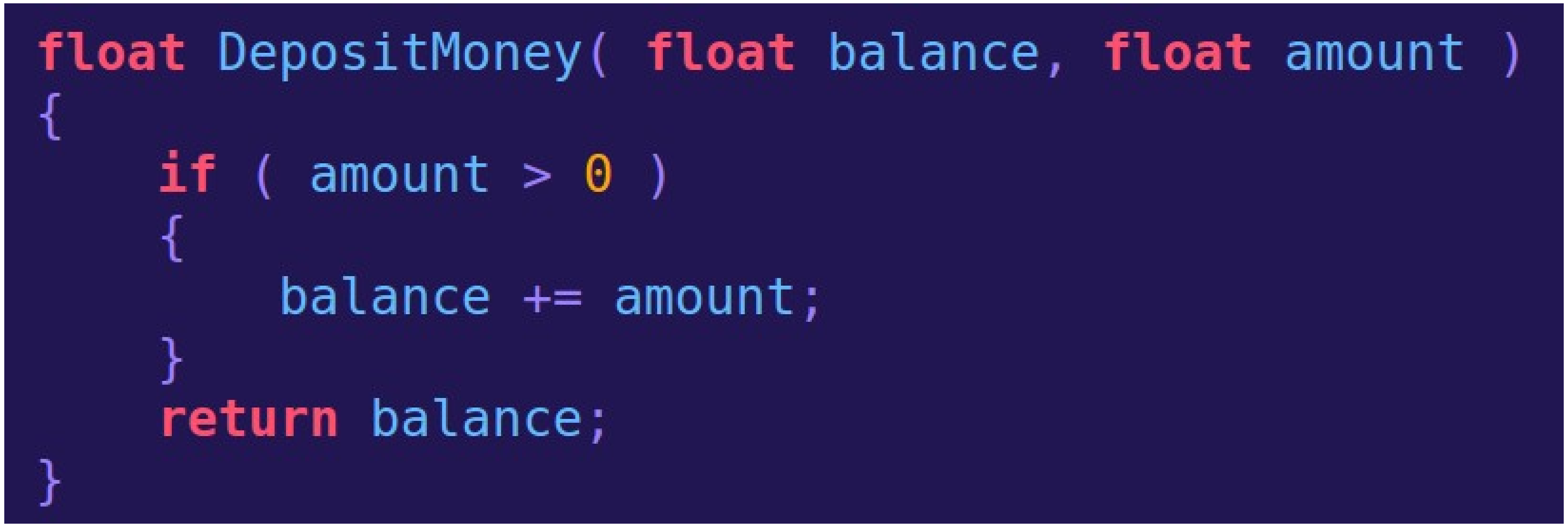


Anatomy of a Function return types

If we want to create a function that does **not** return anything,
We use a return type of **void**. Then, no **`return`** statement is required.



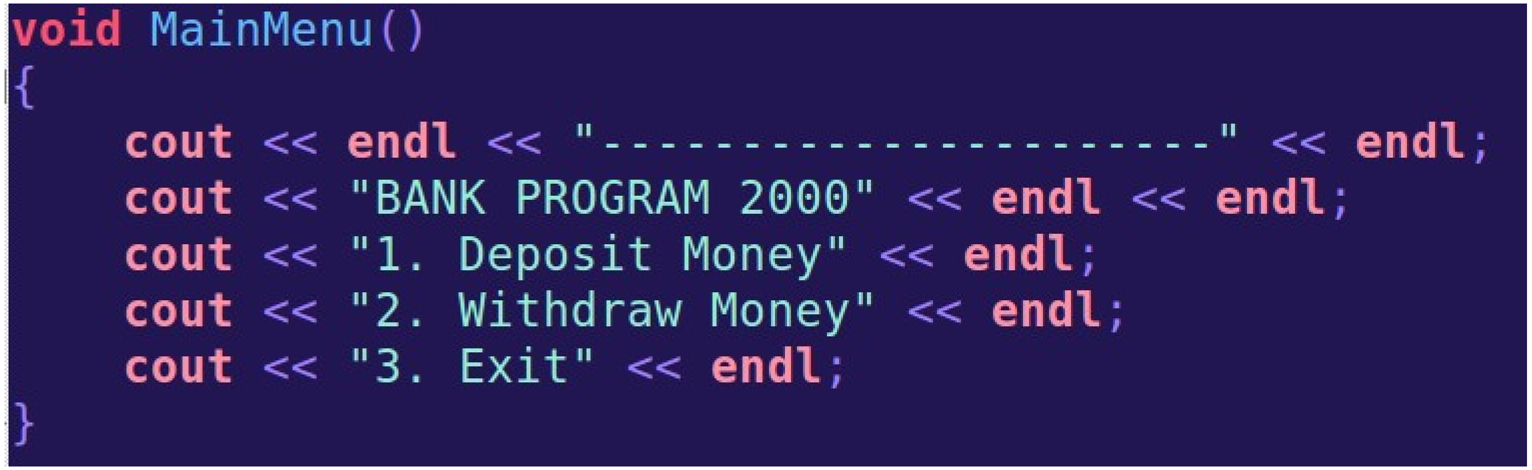
```
void SayHello()
{
    cout << "Hello, World!" << endl;
}
```



```
float DepositMoney( float balance, float amount )
{
    if ( amount > 0 )
    {
        balance += amount;
    }
    return balance;
}
```

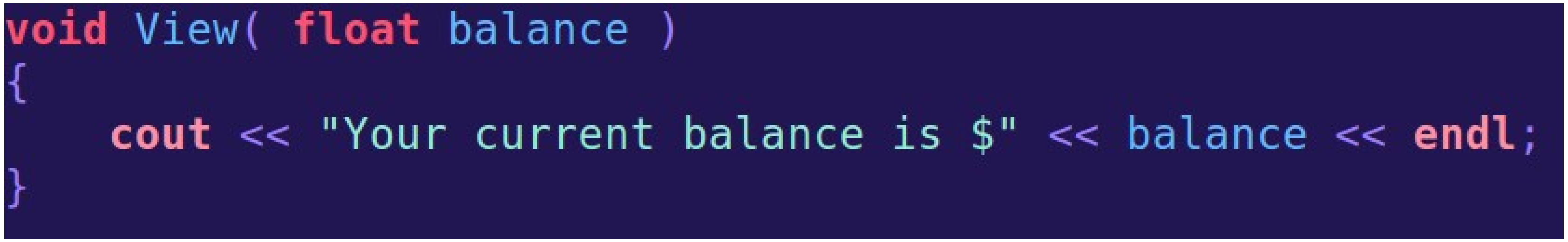
Anatomy of a Function
return types

For example, we could store our main menu within a **void** function,
And display it each time through the program.



```
void MainMenu()
{
    cout << endl << "-----------------------" << endl;
    cout << "BANK PROGRAM 2000" << endl << endl;
    cout << "1. Deposit Money" << endl;
    cout << "2. Withdraw Money" << endl;
    cout << "3. Exit" << endl;
}
```

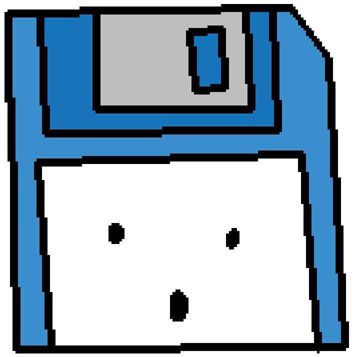
Or just format some information before displaying it.



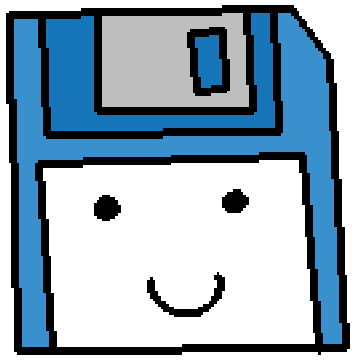
```
void View( float balance )
{
    cout << "Your current balance is $" << balance << endl;
}
```

That was a lot to cover, but we're not done yet!

Let's do some programming before we continue!



Scope

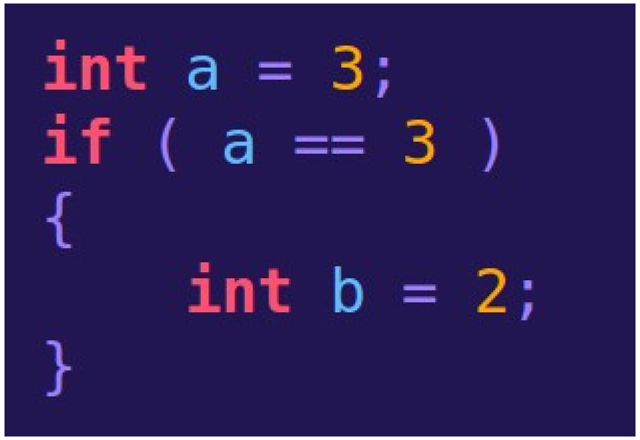


What is Scope?

Scope refers to **where** a variable or function is available within a program.

A variable **declared** within a **block of code** is only accessible within that block.

Think of a **block of code** as being within curly-braces { }
Such as within an if statement, loop, or function body.



```
int a = 3;
if ( a == 3 )
{
    int b = 2;
}
```

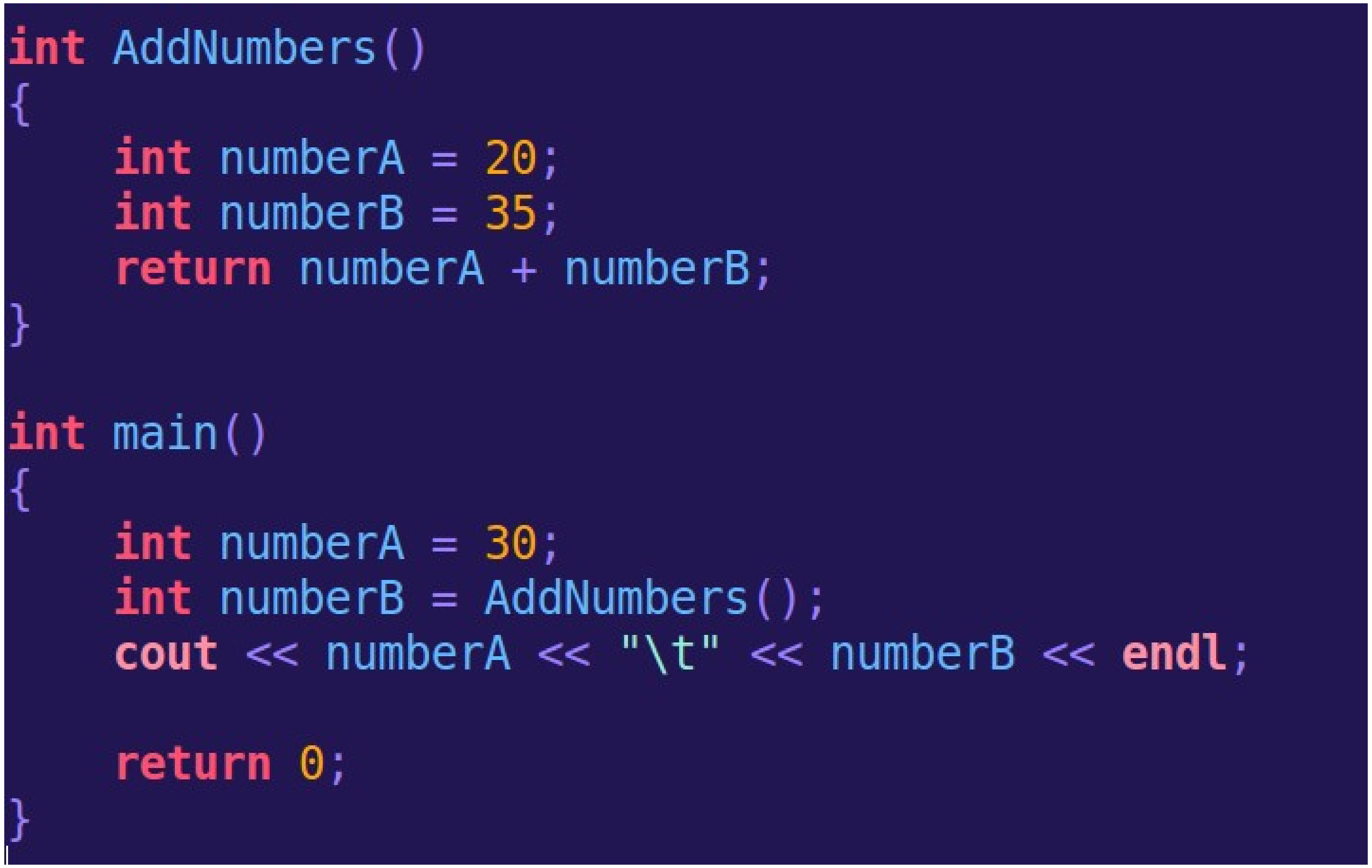
b only exists within this if statement.

Outside of the if statement, it does not exist and cannot be used,

Though a **different** variable with the same name could be created.

What is Scope?

Variable names can be reused if the **scope** is different for both.



```
int AddNumbers()
{
    int numberA = 20;
    int numberB = 35;
    return numberA + numberB;
}

int main()
{
    int numberA = 30;
    int numberB = AddNumbers();
    cout << numberA << "\t" << numberB << endl;

    return 0;
}
```

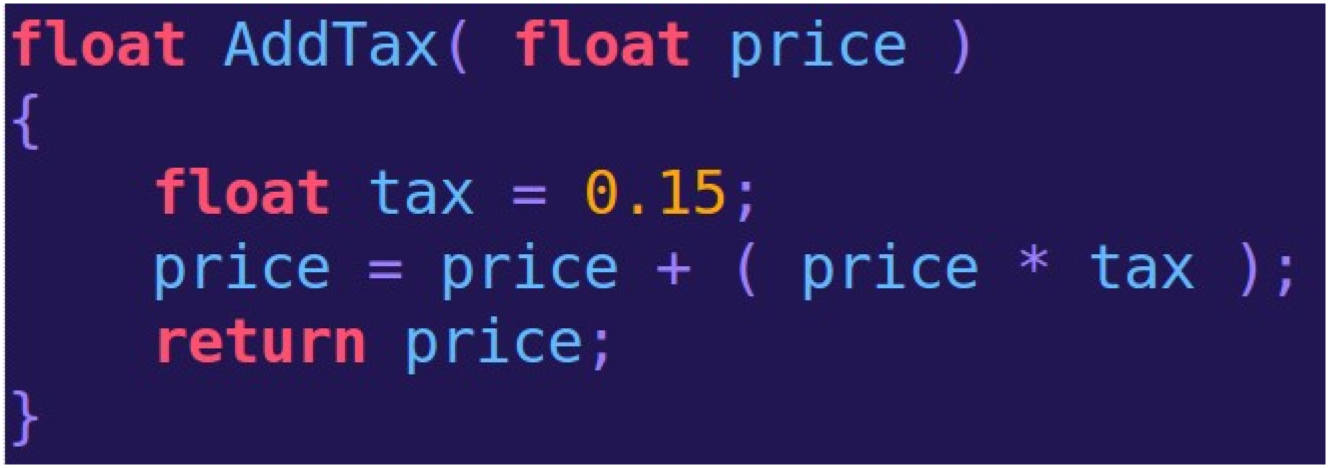
`numberA` and **`numberB`** here are within the scope of the **`AddNumbers`** function.

`numberA` and **`numberB`** here are within the scope of the **`main`** function.

Between each function, they are totally different variables, but with the same name.

What is Scope?

A variable that is within the scope of some code block is known as a **local variable**. These variables are only accessible from within this **scope**.

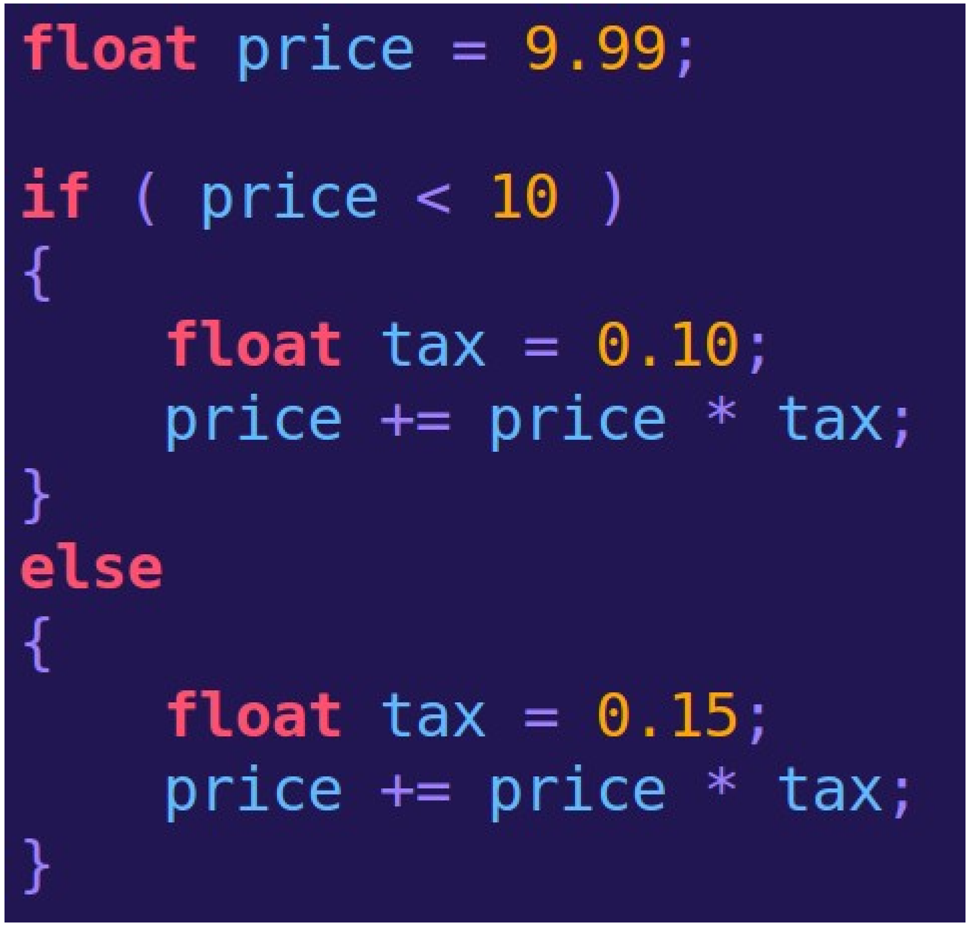


```
float AddTax( float price )
{
    float tax = 0.15;
    price = price + ( price * tax );
    return price;
}
```

`tax` is a local variable of **`AddTax`**.

What is Scope?

A variable that is within the scope of some code block is known as a **local variable**. These variables are only accessible from within this **scope**.



```
float price = 9.99;

if ( price < 10 )
{
    float tax = 0.10;
    price += price * tax;
}
else
{
    float tax = 0.15;
    price += price * tax;
}
```

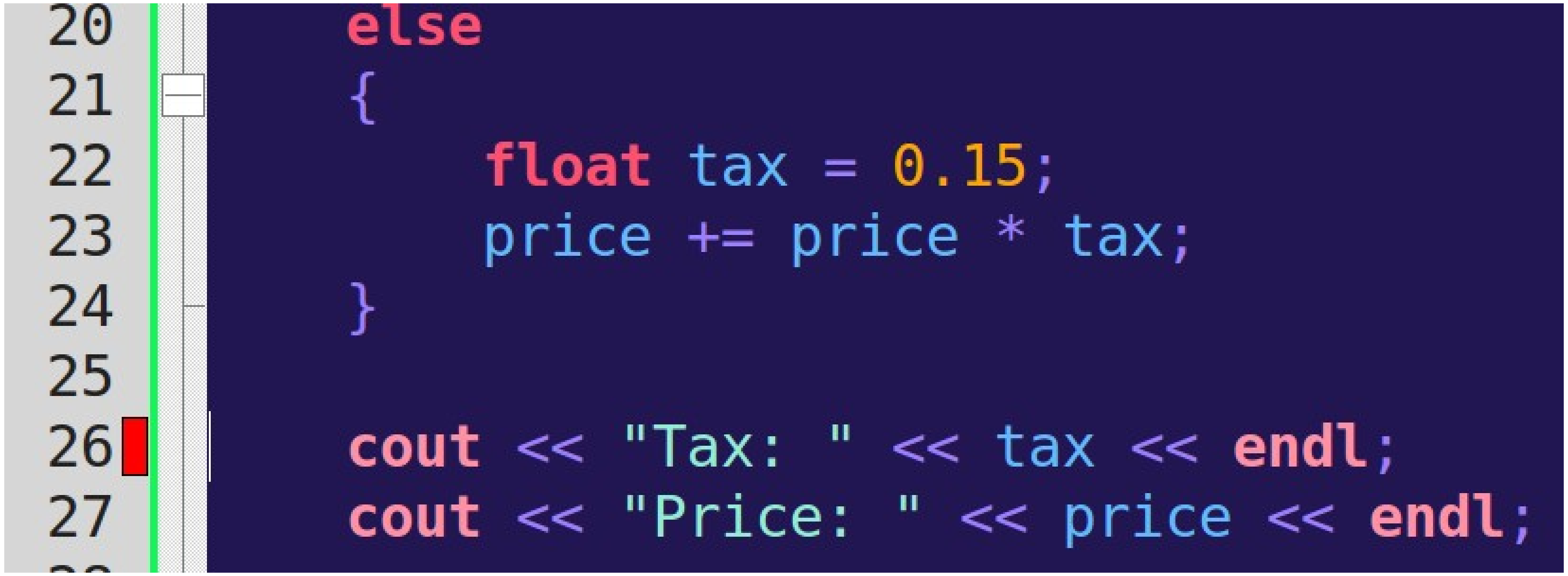
There are two separate variables named **`tax`**, and they have different **scope**.

One is **local** to
`if ( price < 10 )`

And the other is **local** to
`else`

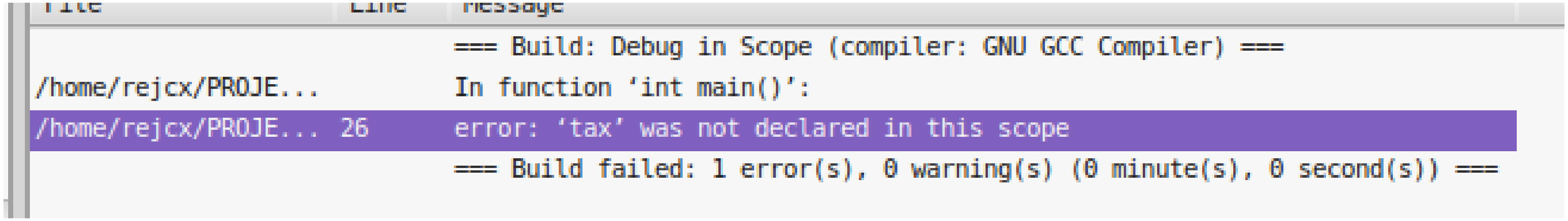
And neither is **in-scope** (neither exists) outside of this if and else statement.

What is Scope?



```
20        else
21        {
22            float tax = 0.15;
23            price += price * tax;
24        }
25
26        cout << "Tax: " << tax << endl;
27        cout << "Price: " << price << endl;
```

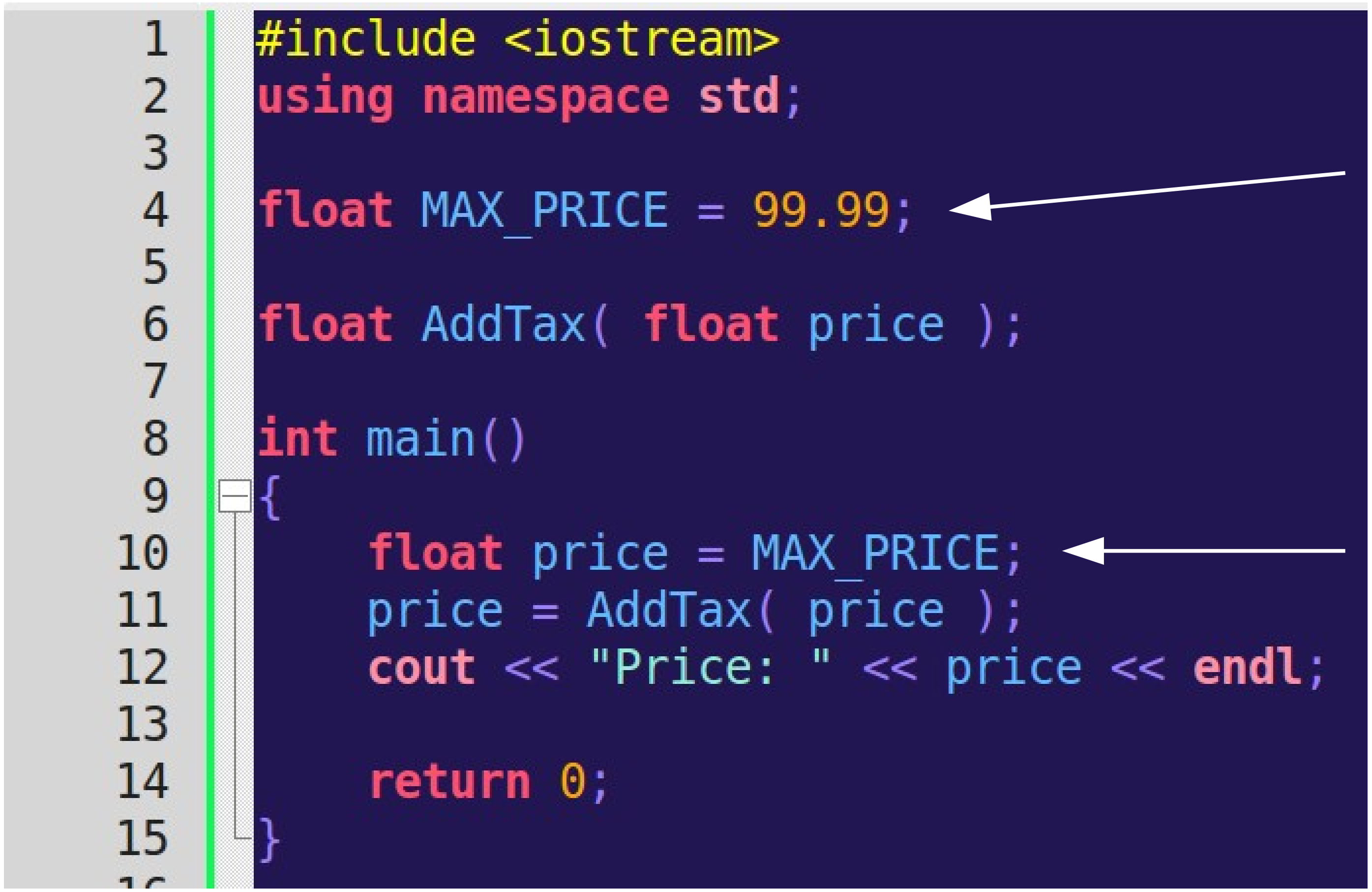
If we try to use a **local variable** outside of its **scope**, we will get a **build error** (a **compile-time** error), because the variable <u>does not exist</u>.



```
                                    === Build: Debug in Scope (compiler: GNU GCC Compiler) ===
/home/rejcx/PROJE...                In function 'int main()':
/home/rejcx/PROJE...  26            error: 'tax' was not declared in this scope
                                    === Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Global Variables

A **global variable** is declared outside of all functions – including the **`main()`** function. A global variable is accessible to all code below it.



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

float MAX_PRICE = 99.99;

float AddTax( float price );

int main()
{
    float price = MAX_PRICE;
    price = AddTax( price );
    cout << "Price: " << price << endl;

    return 0;
}
```

Global variable

Using the global variable

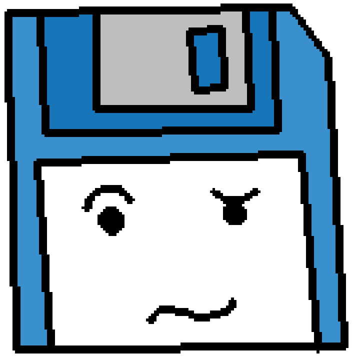
Global Variables

Global variables are considered **bad practice**, and you should avoid them!
Using global variables can lead to...

Hard-to-follow code!

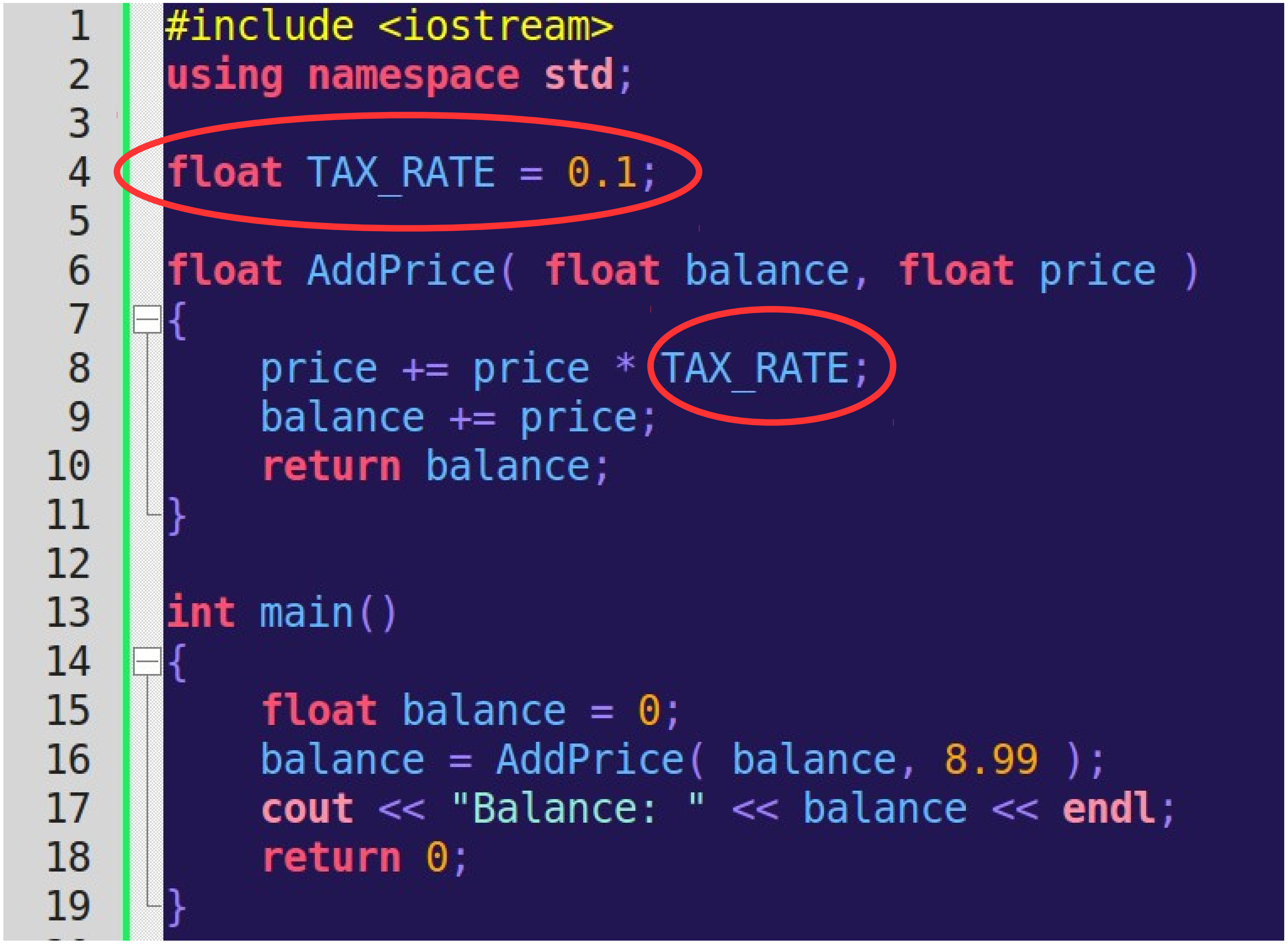
Hard-to-maintain code!

A huge mess!



If you want a variable to be accessible between different functions, you should be **passing the variable as an argument!**

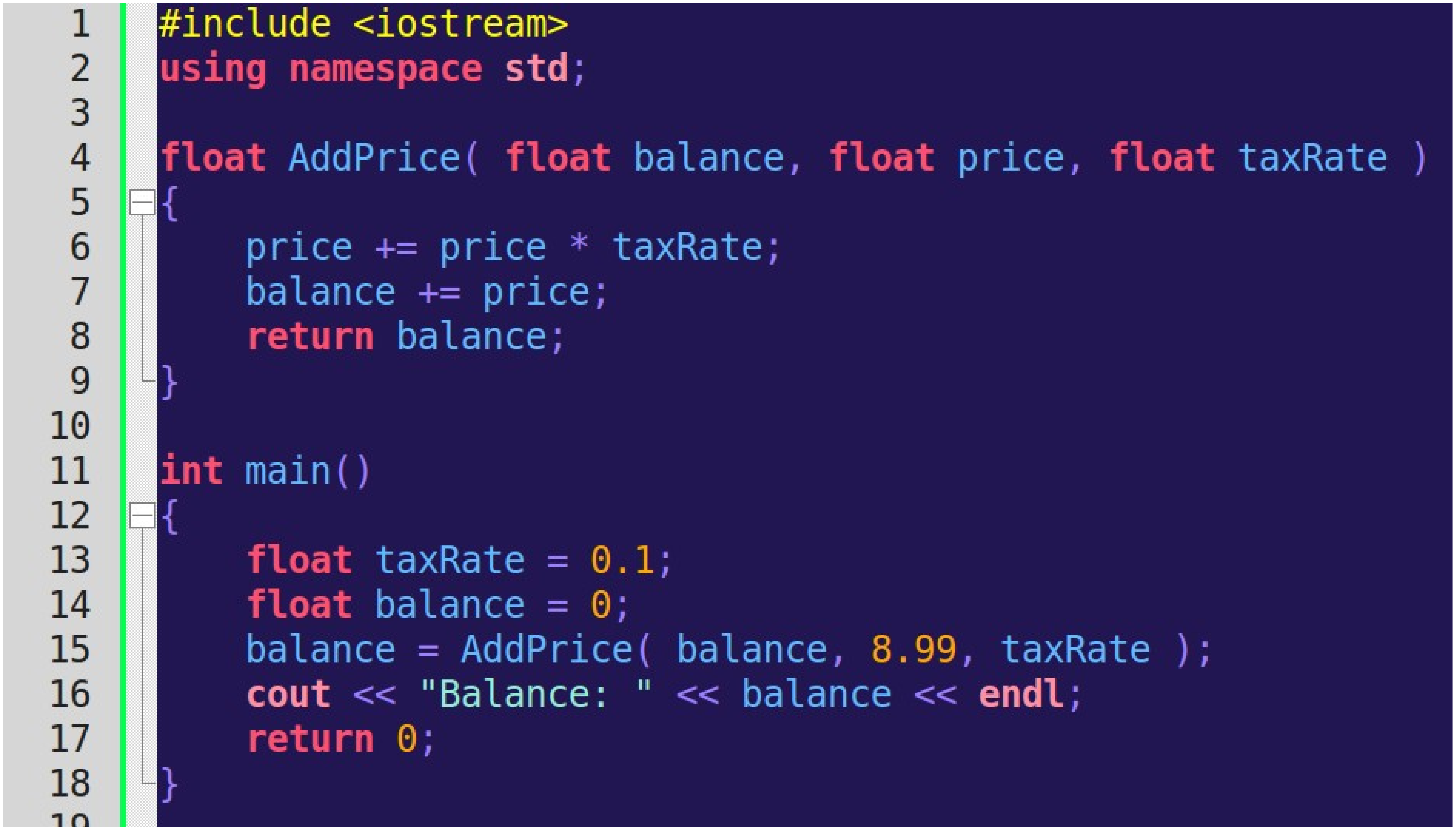
Global Variables



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

float TAX_RATE = 0.1;

float AddPrice( float balance, float price )
{
    price += price * TAX_RATE;
    balance += price;
    return balance;
}

int main()
{
    float balance = 0;
    balance = AddPrice( balance, 8.99 );
    cout << "Balance: " << balance << endl;
    return 0;
}
```

Bad Practice

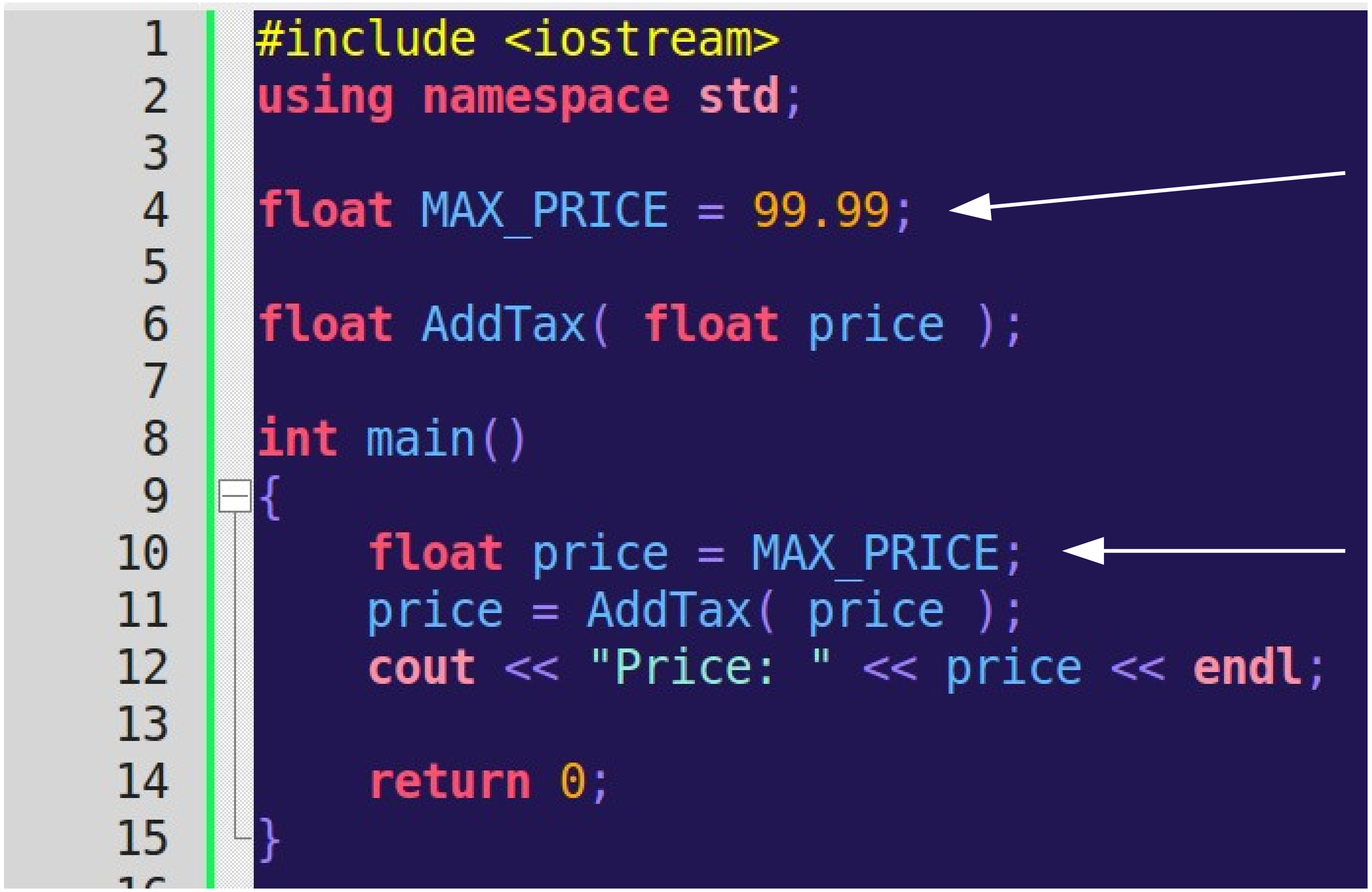
Global Variables


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

float AddPrice( float balance, float price, float taxRate )
{
    price += price * taxRate;
    balance += price;
    return balance;
}

int main()
{
    float taxRate = 0.1;
    float balance = 0;
    balance = AddPrice( balance, 8.99, taxRate );
    cout << "Balance: " << balance << endl;
    return 0;
}
```

Better!

Global Variables

A **global variable** is declared outside of all functions – including the **`main()`** function. A global variable is accessible to all code below it.

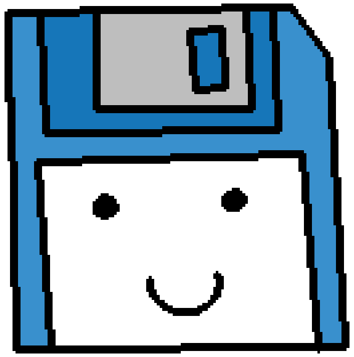


```
#include <iostream>
using namespace std;

float MAX_PRICE = 99.99;

float AddTax( float price );

int main()
{
    float price = MAX_PRICE;
    price = AddTax( price );
    cout << "Price: " << price << endl;

    return 0;
}
```

Global variable

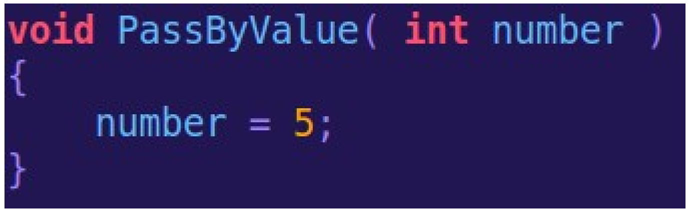
Using the global variable

Passing Parameters



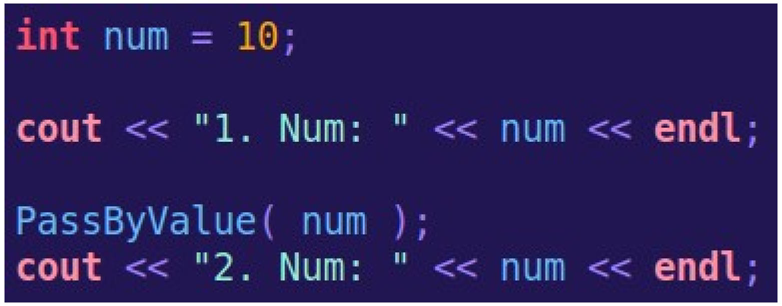
Passing Parameters

When we **pass** a variable to a function, a **copy is made** by default.



```
void PassByValue( int number )
{
    number = 5;
}
```

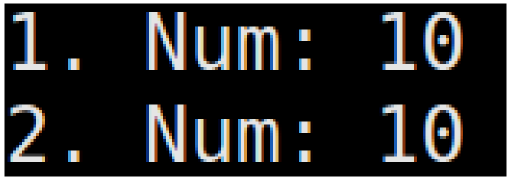
Any changes made within this function to that variable do not get saved once the variable leaves the function (because it's a copy – not the original!)



```
int num = 10;

cout << "1. Num: " << num << endl;

PassByValue( num );
cout << "2. Num: " << num << endl;
```

This is called **passing by value**.

We are passing the **value** of a variable to the function, so that the function can work with it.



```
1. Num: 10
2. Num: 10
```

When the program **returns** from the function
(via a **`return`** statement or just reaching the end of the function)
The variable will still have its original value.

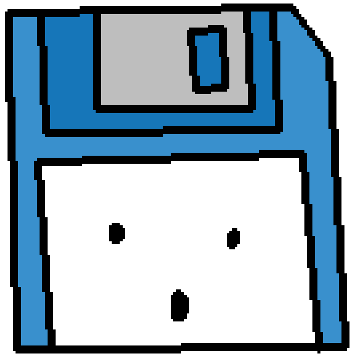
Passing Parameters

We may want a **variable to change** within a function.

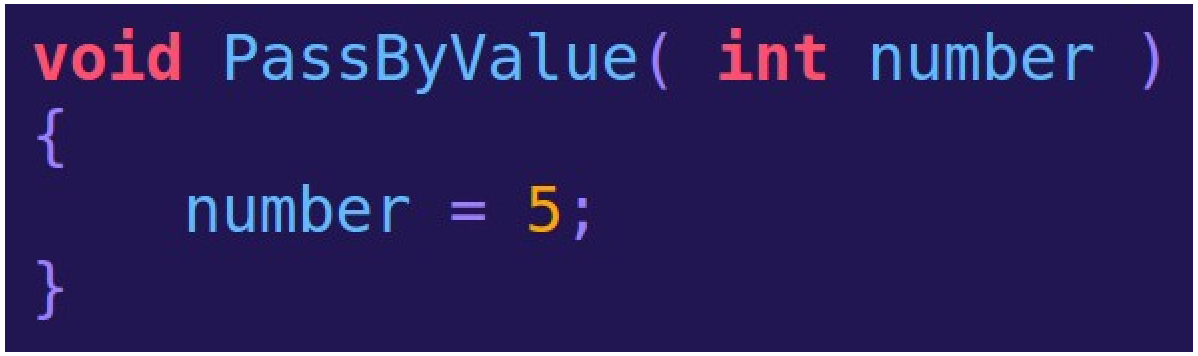
Or, we may want to **return multiple values** from the function.

In C++, we can only use **`return`** to return ONE value,

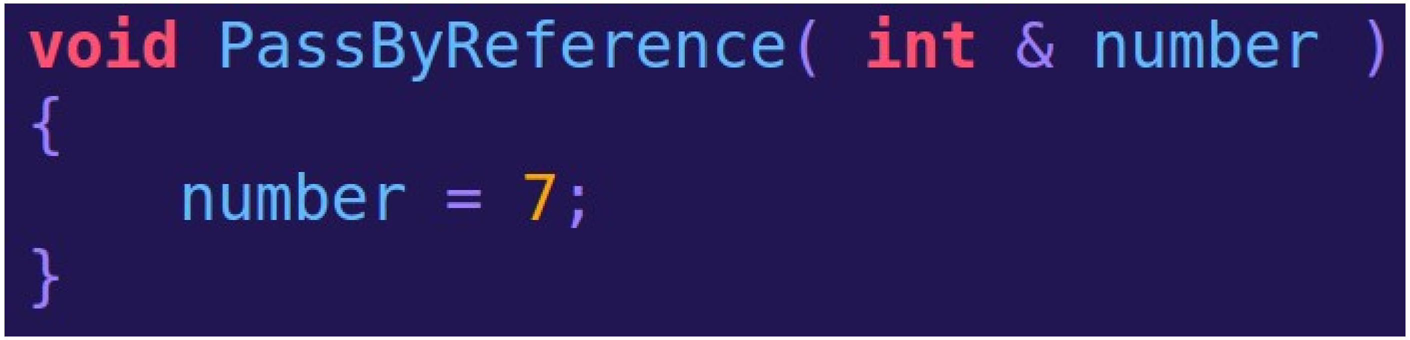
But there is another way we can return information from a function.



Passing Parameters



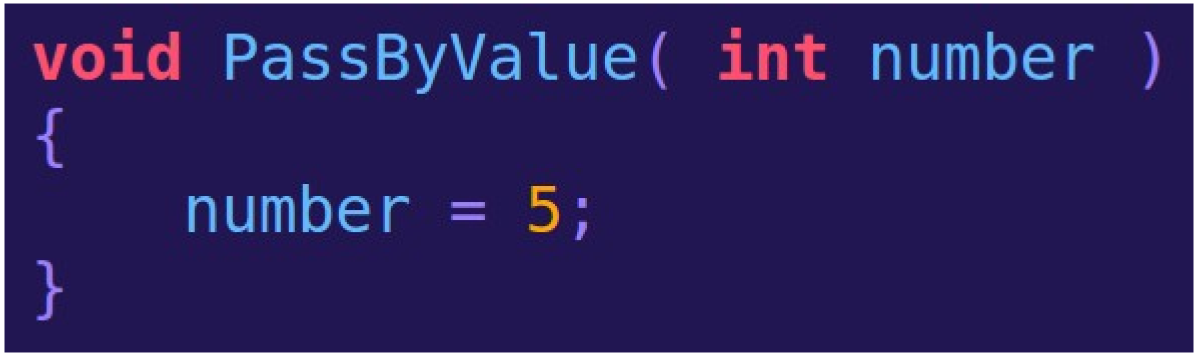
```
void PassByValue( int number )
{
    number = 5;
}
```



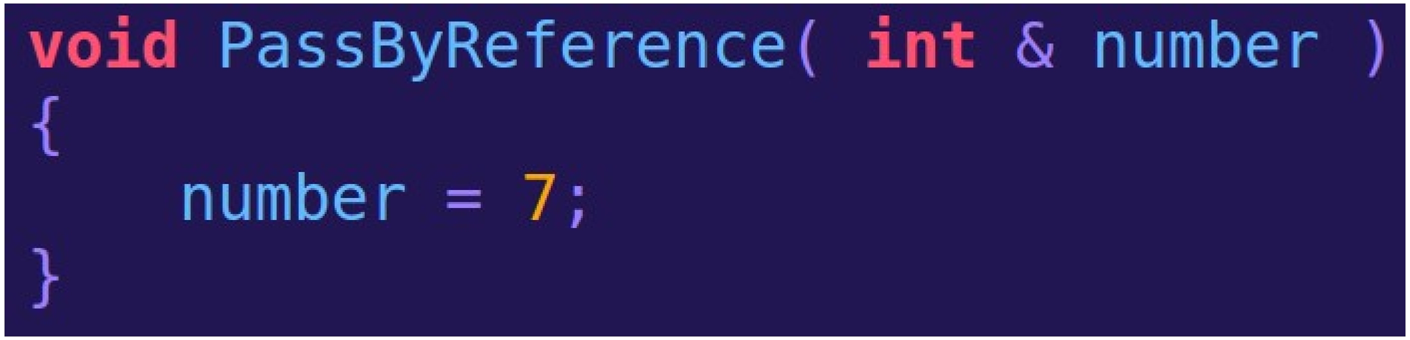
```
void PassByReference( int & number )
{
    number = 7;
}
```

If we want to pass our arguments by **reference**, within our **function header**, we need to specify those **parameters** with an ampersand & after the **data-type**.

Passing Parameters



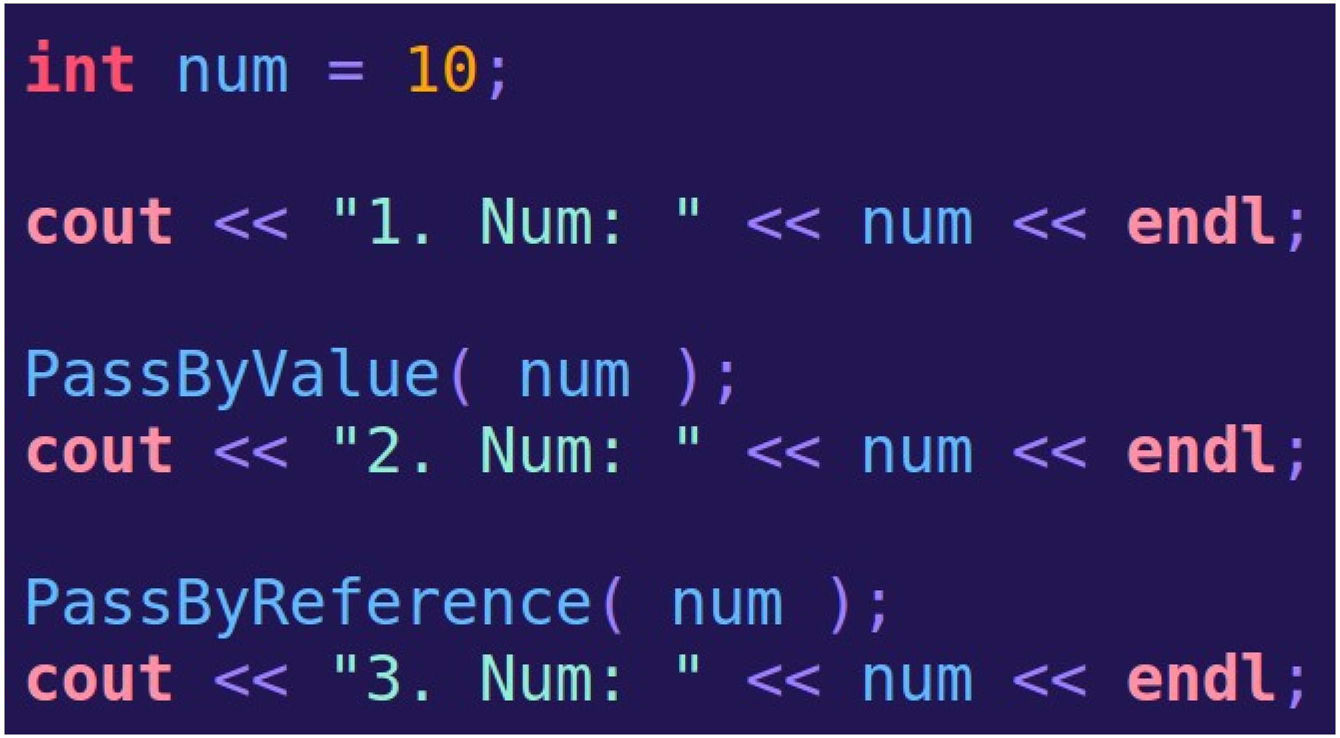
```
void PassByValue( int number )
{
    number = 5;
}
```



```
void PassByReference( int & number )
{
    number = 7;
}
```

This is the only thing that changes when passing **by reference**. We still use our **number** variable the same way within the **function body**.

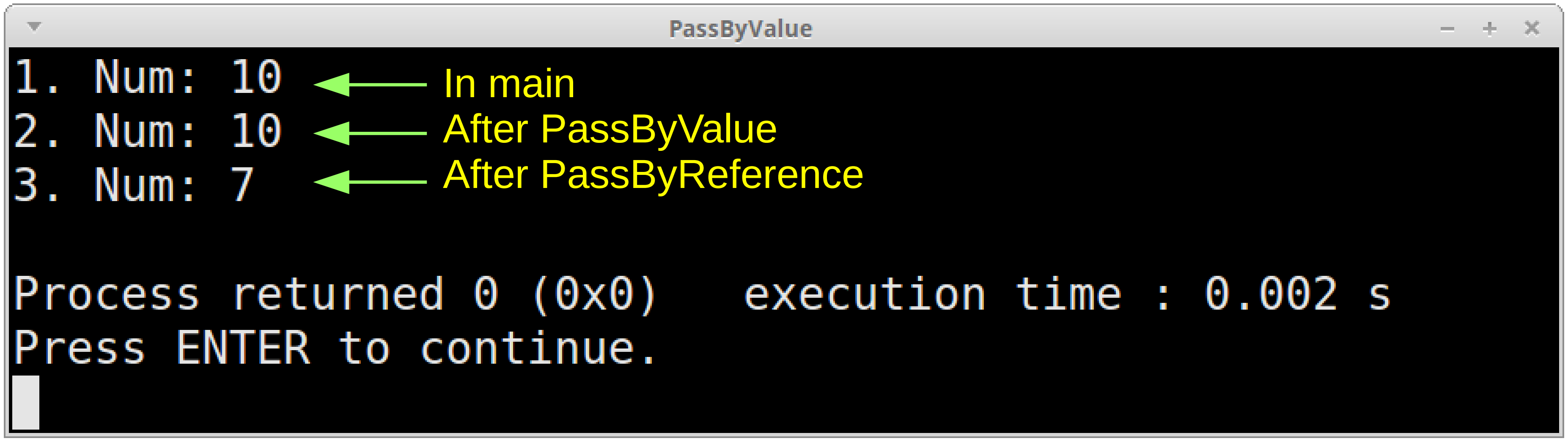
Passing Parameters



```
int num = 10;

cout << "1. Num: " << num << endl;

PassByValue( num );
cout << "2. Num: " << num << endl;

PassByReference( num );
cout << "3. Num: " << num << endl;
```

We still **call our function** in the same way.

Passing Parameters

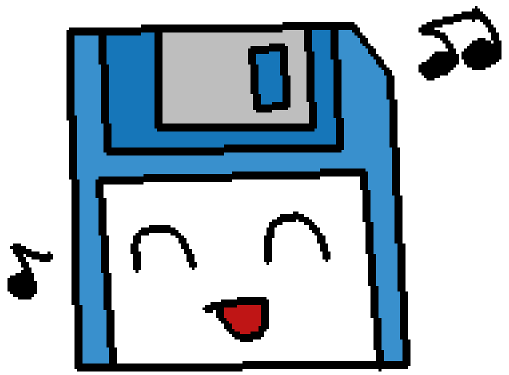


```
1. Num: 10
2. Num: 10
3. Num: 7

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

But now the variable that we **pass into the function** during the **function call** gets updated.

Passing Parameters

Let's do some coding to put it into practice!



Passing Parameters

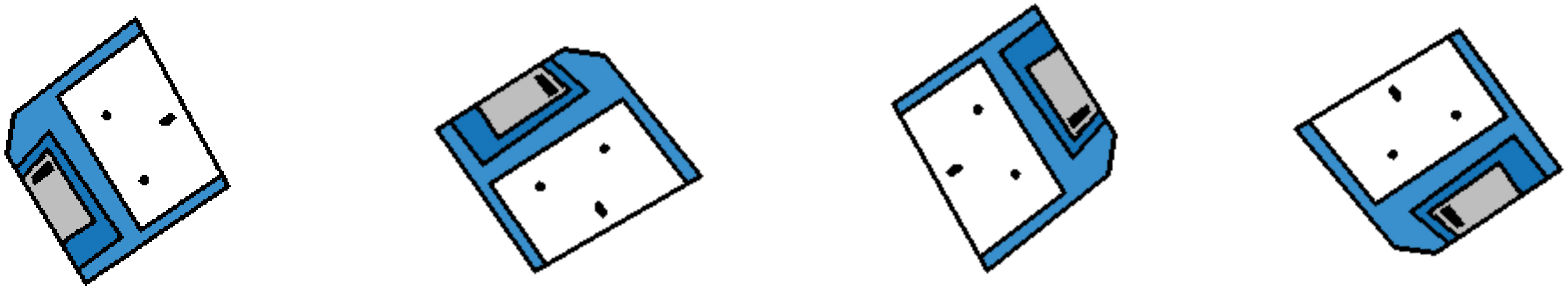
So remember

Pass by Value

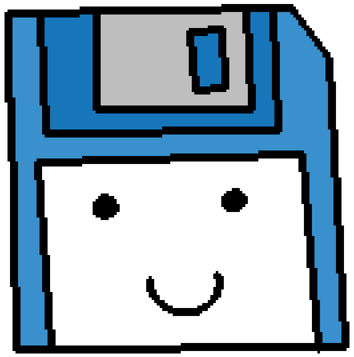
And

Pass by Reference

There are a couple more things to cover! Hold tight!

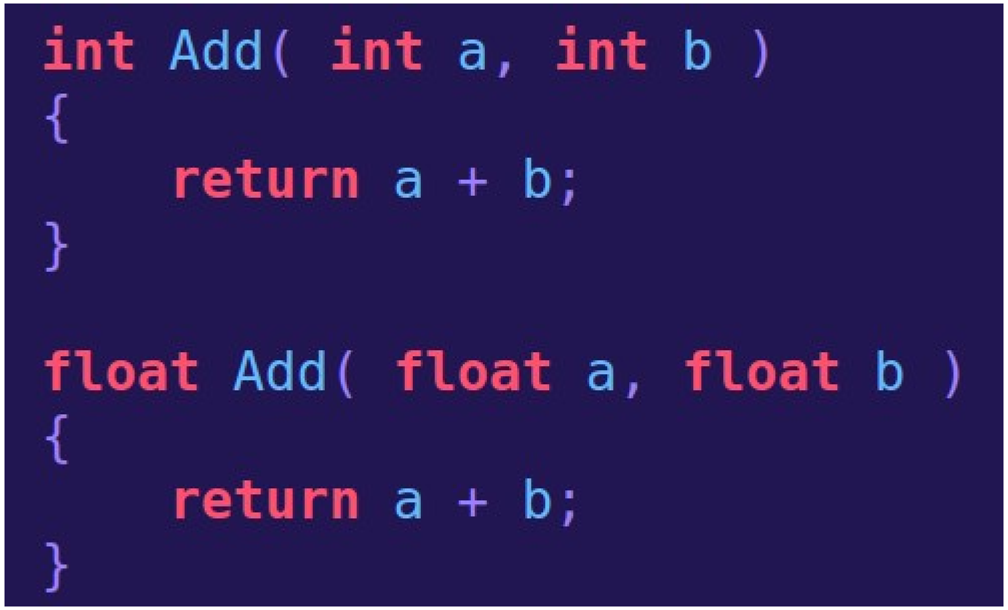


Function Overloading



Function Overloading

In C++, we can **reuse function names**.



```
int Add( int a, int b )
{
    return a + b;
}

float Add( float a, float b )
{
    return a + b;
}
```

This is called
function overloading.

As long as the **parameter lists** between both functions are different, using the same function name is valid.

C++ won't look at the **return type** to tell these functions apart, so these don't matter whether they're the same or different.

Function Overloading

A **function signature** is essentially the part of a function header that C++ uses to tell functions apart – The name, and the parameter list.

```
int F();
int F( int a );
int F( float a );
int F( int a, float b );
int F( float a, int b );
```

You can have as many overloaded functions as you wish.

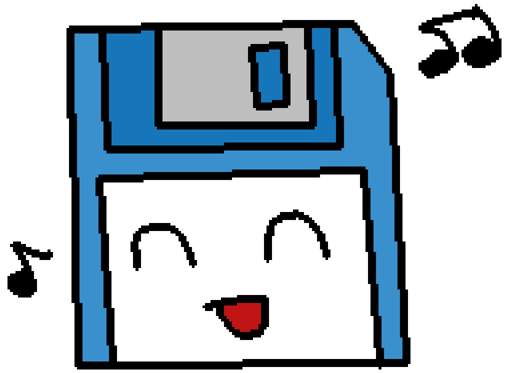
Two functions can be unique if:

- They have a different amount of parameters.
- If the parameters are different data-types.
- If the parameters are in a different order.

C++ doesn't care about the **names** of the parameters. If the parameters all have the same **data-type** then it isn't valid.

Function Overloading

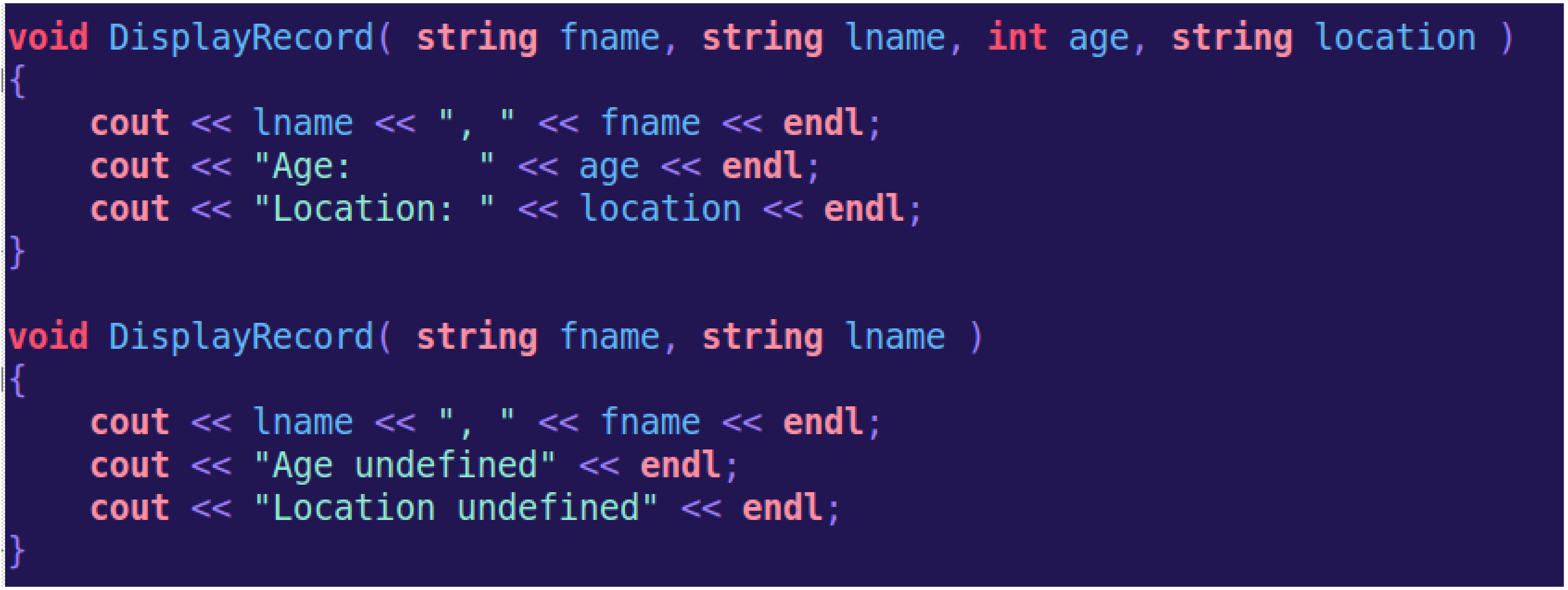
Example!



Default Parameters

A **default parameter** is used to set a **default value** for a function parameter.

For example, we could use overloaded functions to display a record to a user differently, based on what information is available.

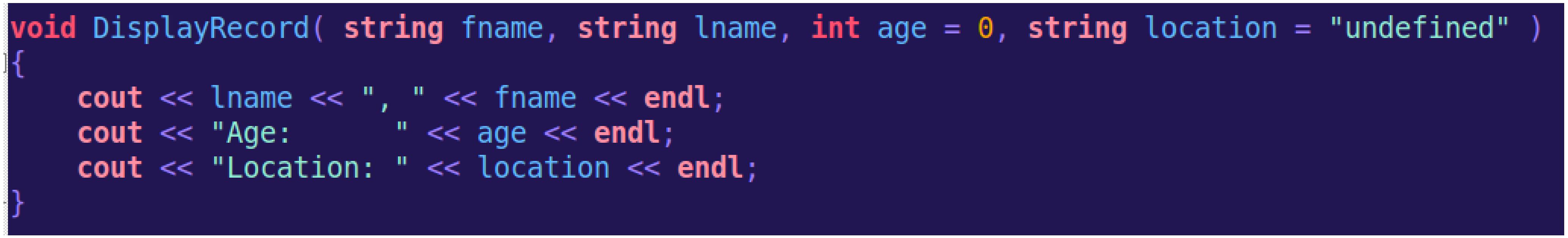


```
void DisplayRecord( string fname, string lname, int age, string location )
{
    cout << lname << ", " << fname << endl;
    cout << "Age:      " << age << endl;
    cout << "Location: " << location << endl;
}

void DisplayRecord( string fname, string lname )
{
    cout << lname << ", " << fname << endl;
    cout << "Age undefined" << endl;
    cout << "Location undefined" << endl;
}
```

But this might get unruly if we want different ways to display records for any amount of data provided.

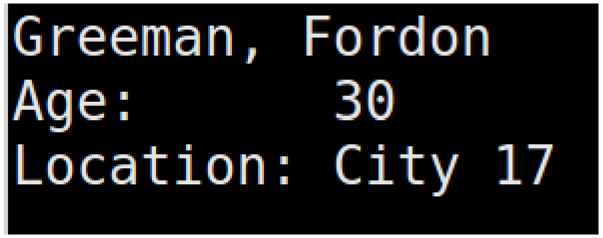
Default Parameters

Instead of overloading this **`DisplayRecord`** function, we could reuse the same code and simply set some **default values**.

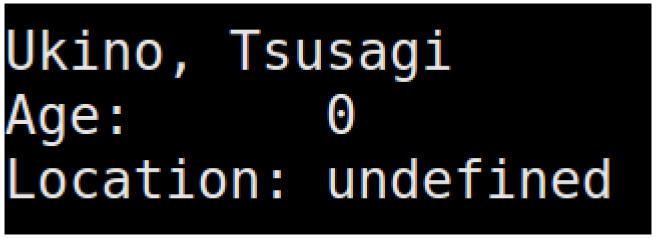


```cpp
void DisplayRecord( string fname, string lname, int age = 0, string location = "undefined" )
{
    cout << lname << ", " << fname << endl;
    cout << "Age:      " << age << endl;
    cout << "Location: " << location << endl;
}
```

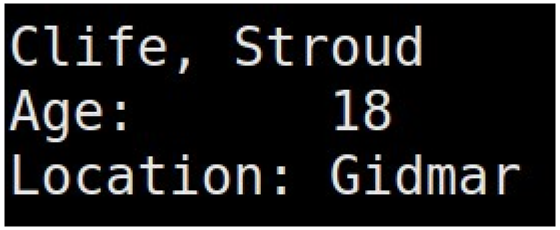
We specify the default value by setting the parameter equal to something. Here, 0 is the default for **`age`** and “undefined” is the default for **`location`**.



```
Greeman, Fordon
Age:      30
Location: City 17
```



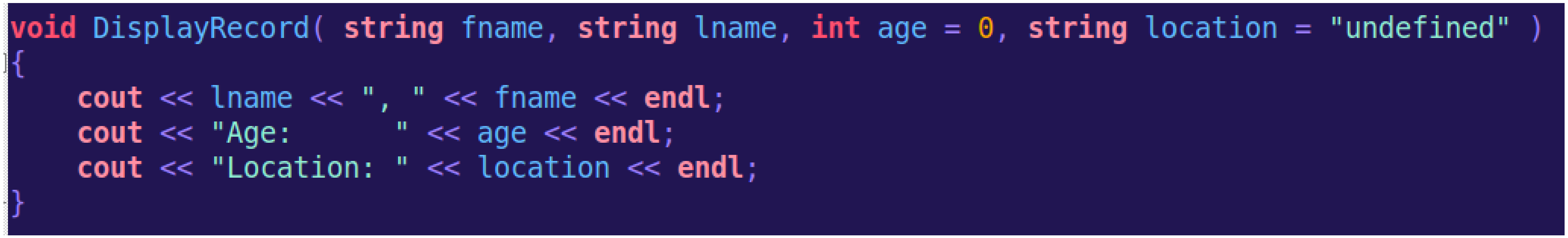
```
Ukino, Tsusagi
Age:      0
Location: undefined
```



```
Clife, Stroud
Age:      18
Location: Gidmar
```

Default Parameters

Instead of overloading this **`DisplayRecord`** function, we could reuse the same code and simply set some **default values**.

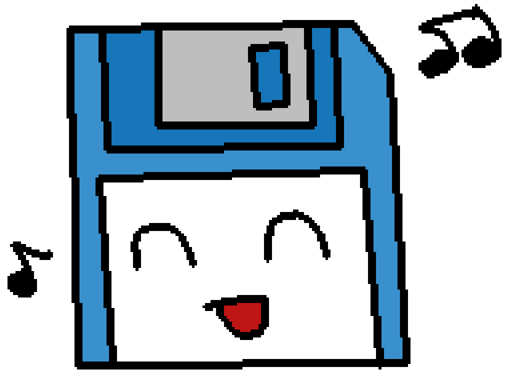


```cpp
void DisplayRecord( string fname, string lname, int age = 0, string location = "undefined" )
{
    cout << lname << ", " << fname << endl;
    cout << "Age:      " << age << endl;
    cout << "Location: " << location << endl;
}
```

The default parameters must go towards the end of the parameter list. There cannot be any **not-default-parameters** that appear after **default parameters**.

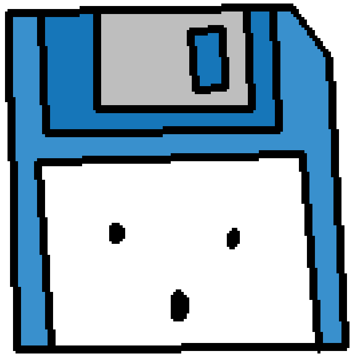
Default Parameters

Example!



That was a lot to cover!

The more you work with functions, overloading, default parameters, and passing by reference, the better you will get with it.



Here are some more sample programs that you can use to get a feel for these topics.