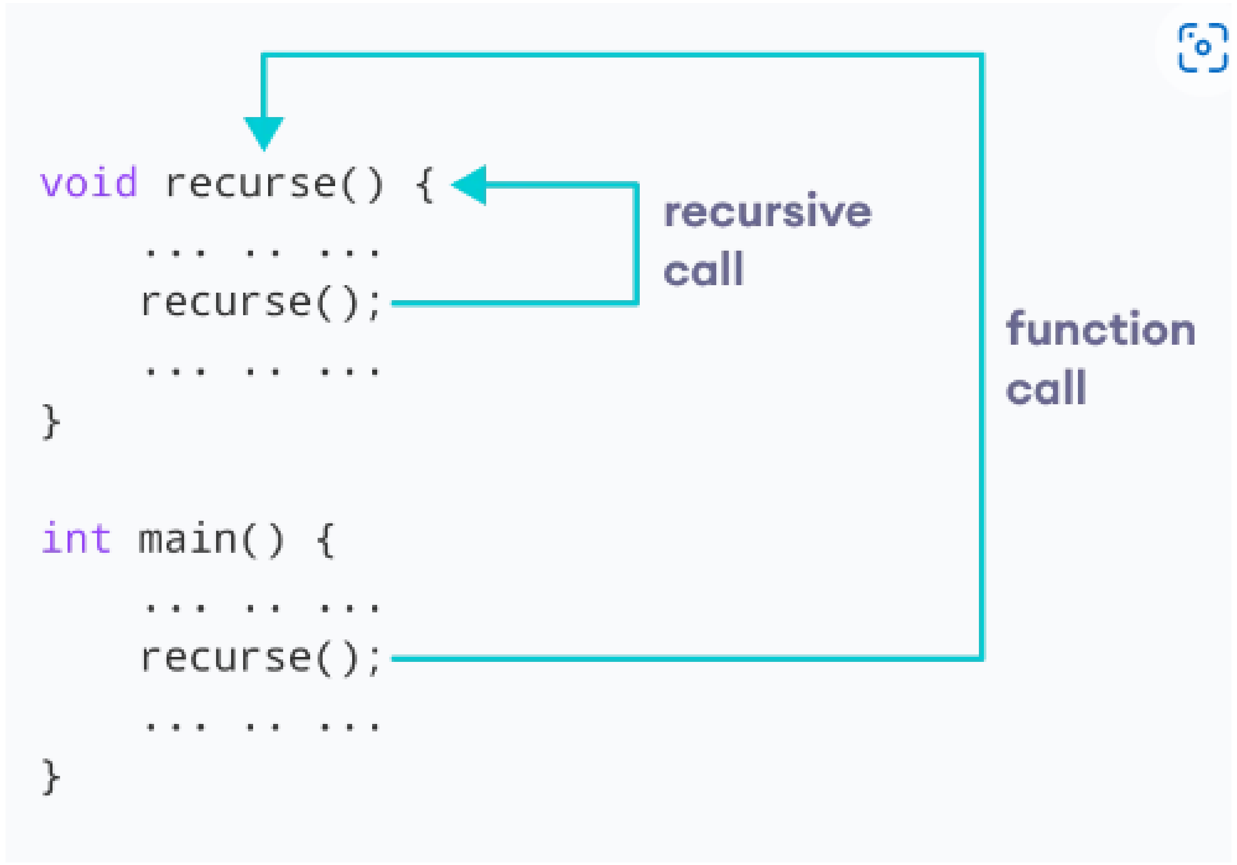
Lecture 13

M M Imran

Recursion

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

```
void recurse() {
  recurse();
}


int main() {
  recurse();
}
```



```
void recurse() {
    ... .. ...
    recurse();
    ... .. ...
}

int main() {
    ... .. ...
    recurse();
    ... .. ...
}
```

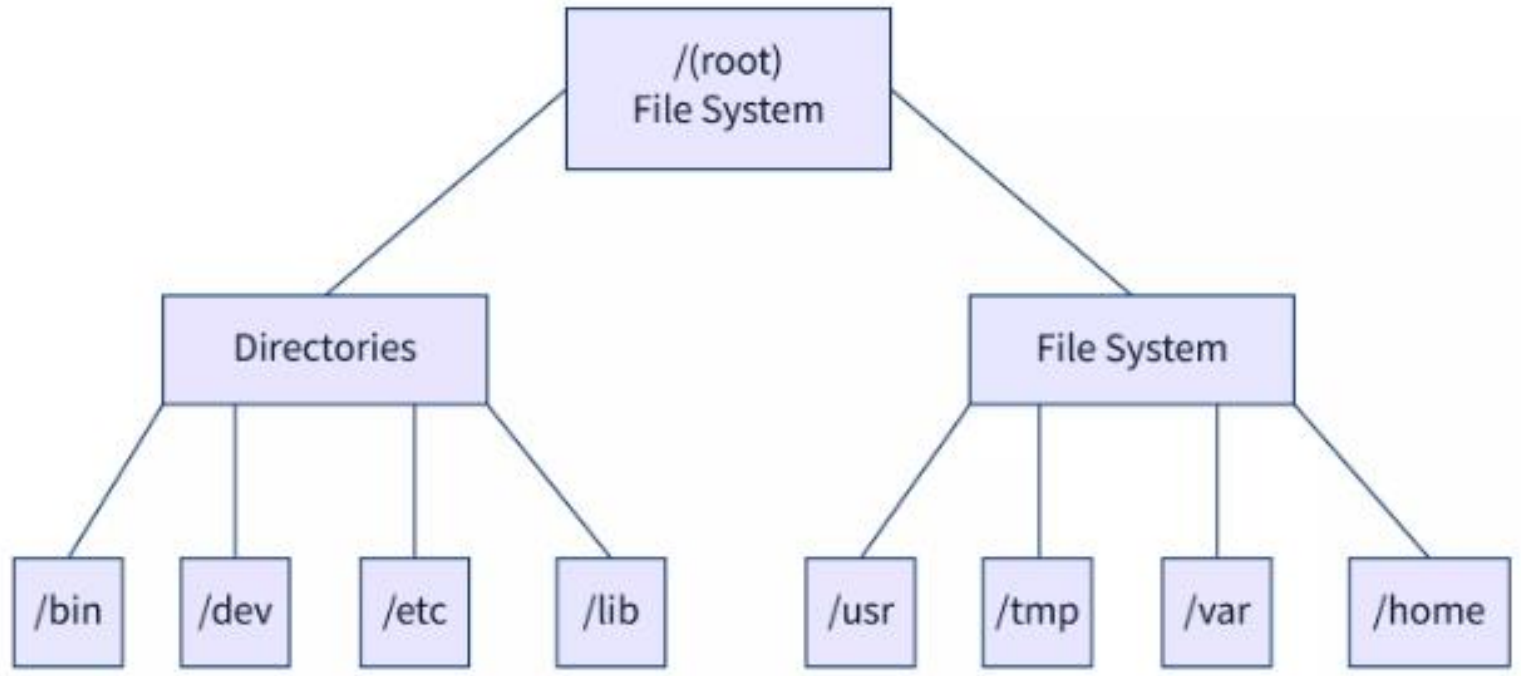
Properties of Recursion

- The recursion continues until some condition is met.
- To prevent infinite recursion, if...else statement (or similar approach) can be used.

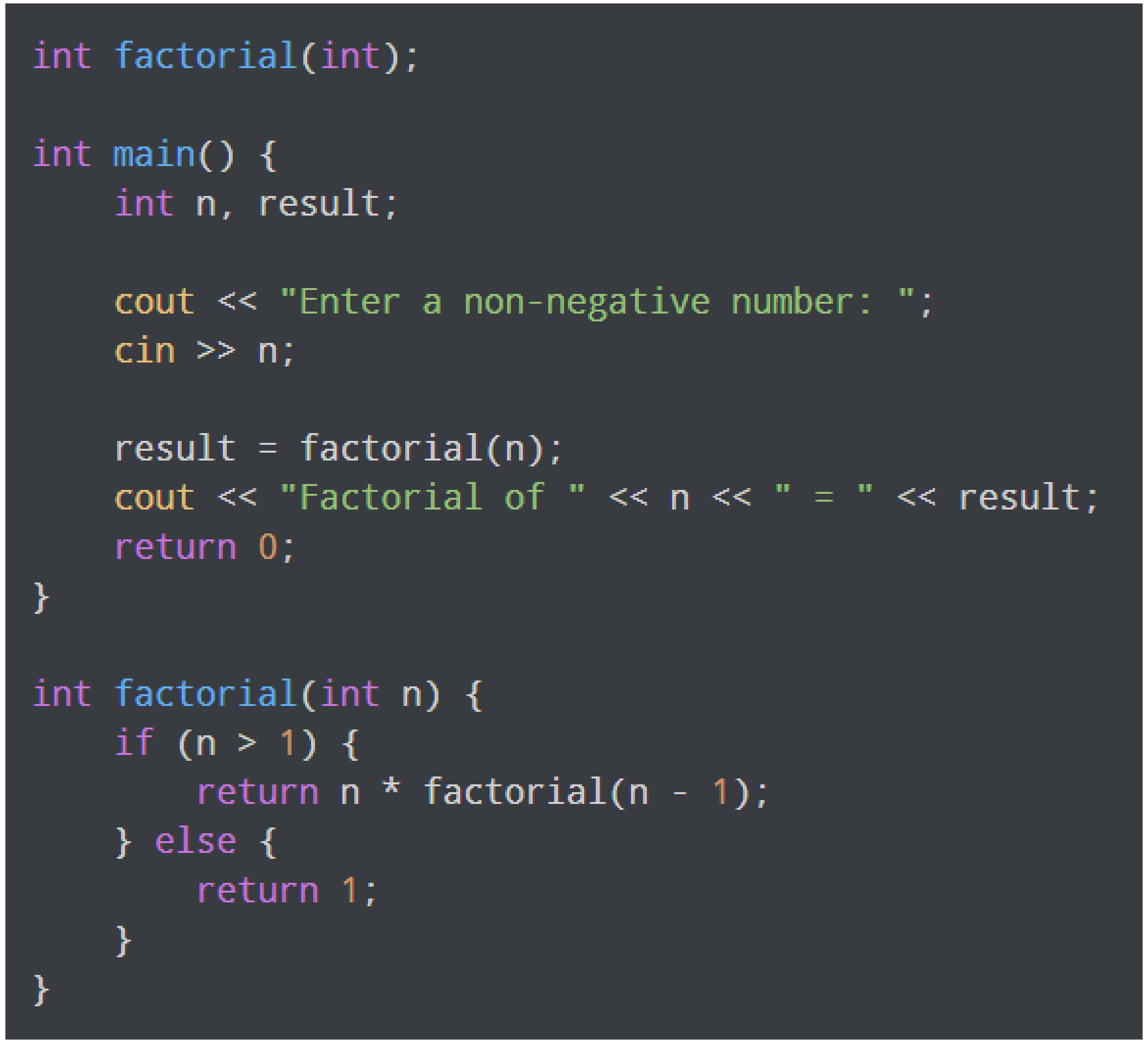
- Sometimes we are unable to solve a problem iteratively due to it's high complexity. But, if we break the problem into smaller versions of the problem, we may be able to find solutions for those smaller versions of the problem. And then, we may combine those solutions to arrive at our answer for the larger problem.

Imagine you are searching for a file on your laptop. You enter the root folder, within that you enter another folder, and so forth as you climb folders on folders until you locate your file.

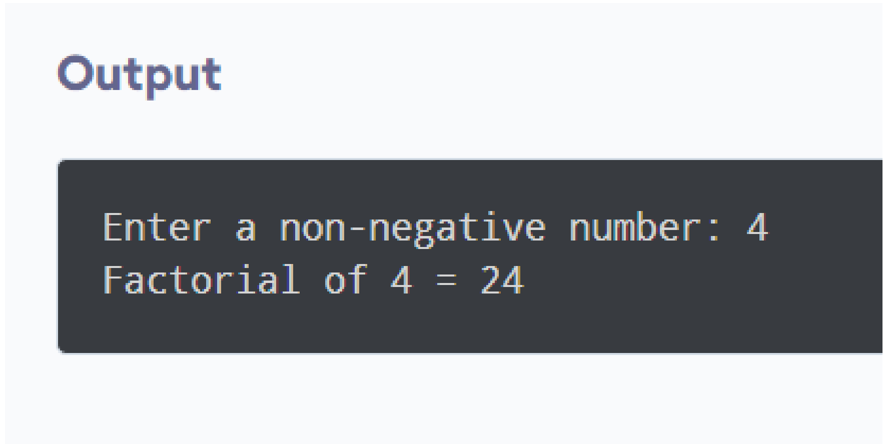
You can think of it as a tree with multiple branches. So, these kinds of problems are best solved using recursion. For these types of problems, an iterative approach may not be preferable.



Factorial of a Number Using Recursion

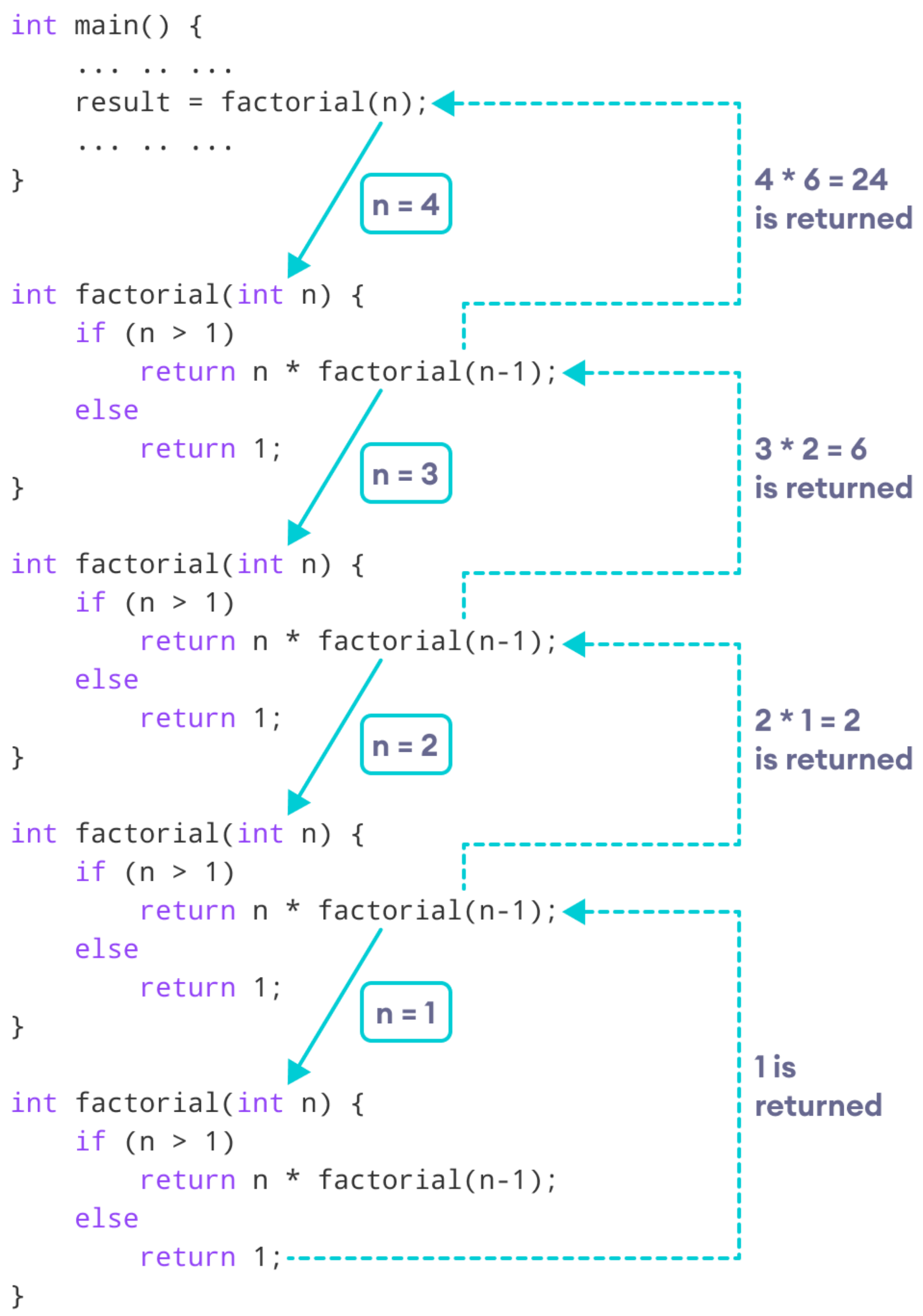


```cpp
int factorial(int);

int main() {
    int n, result;

    cout << "Enter a non-negative number: ";
    cin >> n;

    result = factorial(n);
    cout << "Factorial of " << n << " = " << result;
    return 0;
}

int factorial(int n) {
    if (n > 1) {
        return n * factorial(n - 1);
    } else {
        return 1;
    }
}
```



Output

```
Enter a non-negative number: 4
Factorial of 4 = 24
```



```
int main() {
    ... .. ...
    result = factorial(n);
    ... .. ...
}

int factorial(int n) {
    if (n > 1)
        return n * factorial(n-1);
    else
        return 1;
}

int factorial(int n) {
    if (n > 1)
        return n * factorial(n-1);
    else
        return 1;
}

int factorial(int n) {
    if (n > 1)
        return n * factorial(n-1);
    else
        return 1;
}

int factorial(int n) {
    if (n > 1)
        return n * factorial(n-1);
    else
        return 1;
}
```

Practice

Print from 10 to 1 using recursion

Advantages and Disadvantages of Recursion

Advantages of Recursion

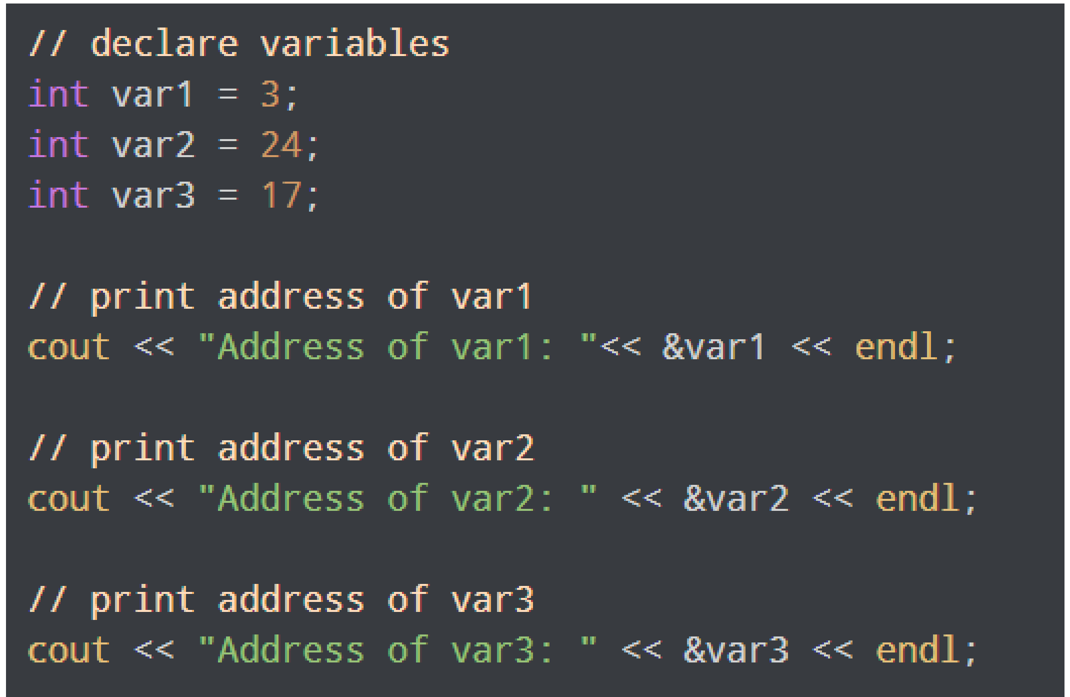
- It makes our code shorter and cleaner.
- Recursion is required in problems concerning data structures and advanced algorithms, such as Graph and Tree Traversal.

Disadvantages of Recursion

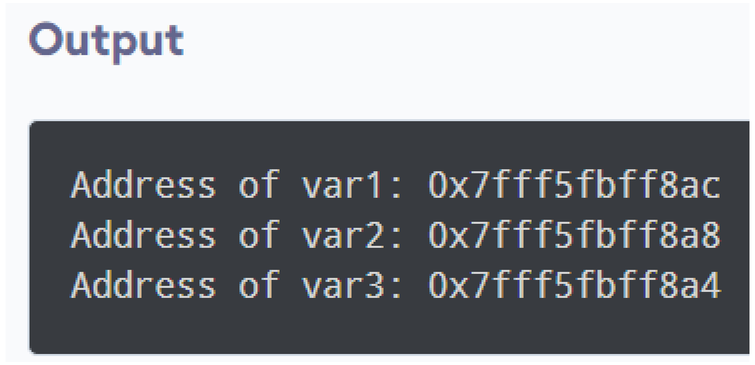
- It takes a lot of stack space compared to an iterative program.
- It uses more processor time.
- It can be more difficult to debug compared to an equivalent iterative program.

Pointer

- In C++, pointers are variables that store the memory addresses of other variables.
- If we have a variable var in our program, &var will give us its address in the memory.



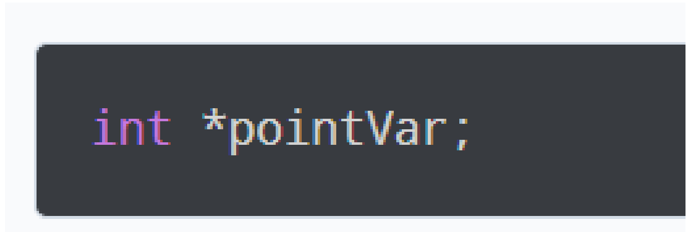
```
// declare variables
int var1 = 3;
int var2 = 24;
int var3 = 17;

// print address of var1
cout << "Address of var1: "<< &var1 << endl;

// print address of var2
cout << "Address of var2: " << &var2 << endl;

// print address of var3
cout << "Address of var3: " << &var3 << endl;
```



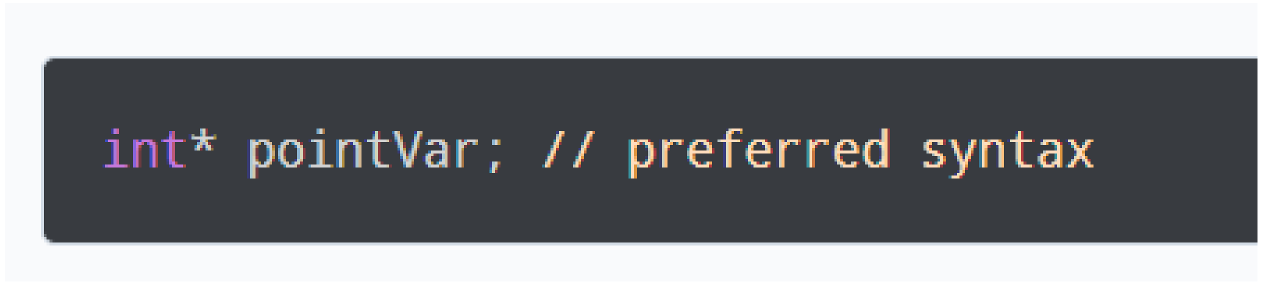
Output

```
Address of var1: 0x7fff5fbff8ac
Address of var2: 0x7fff5fbff8a8
Address of var3: 0x7fff5fbff8a4
```

Syntax

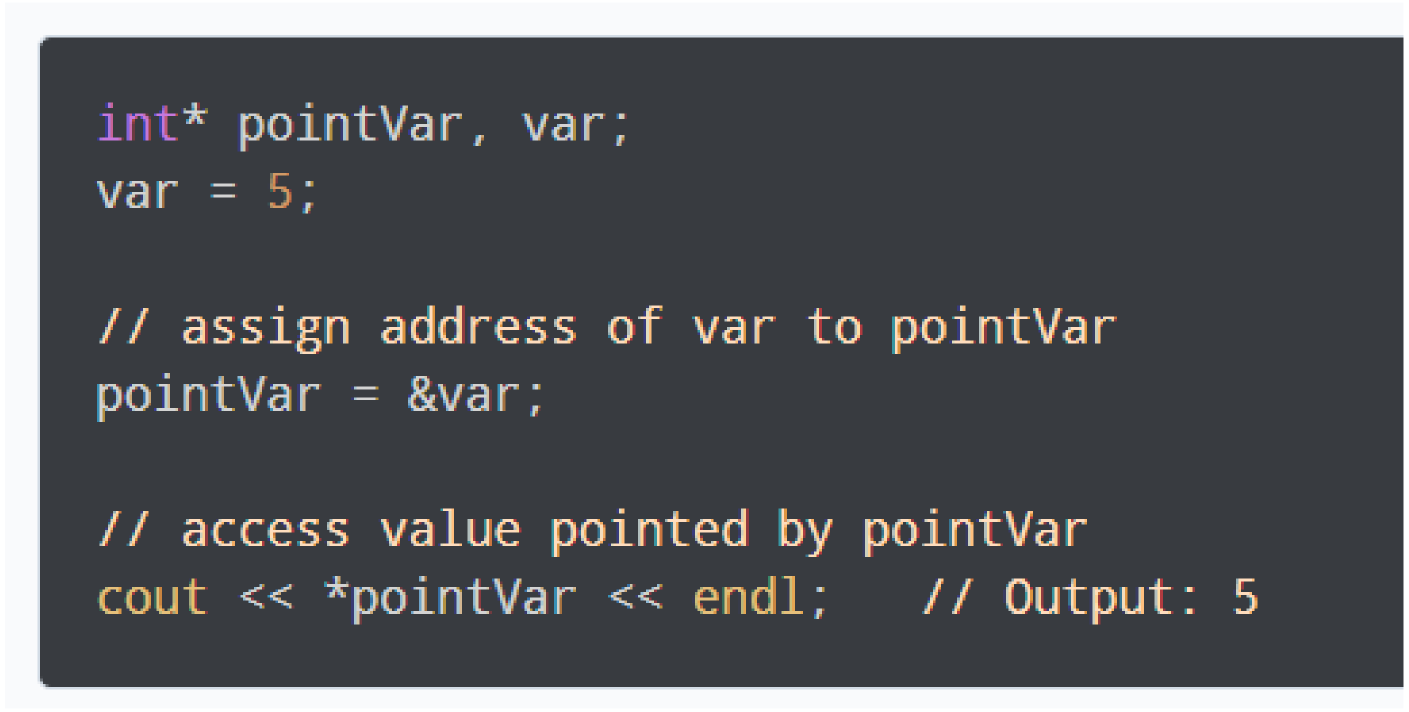


```
int *pointVar;
```

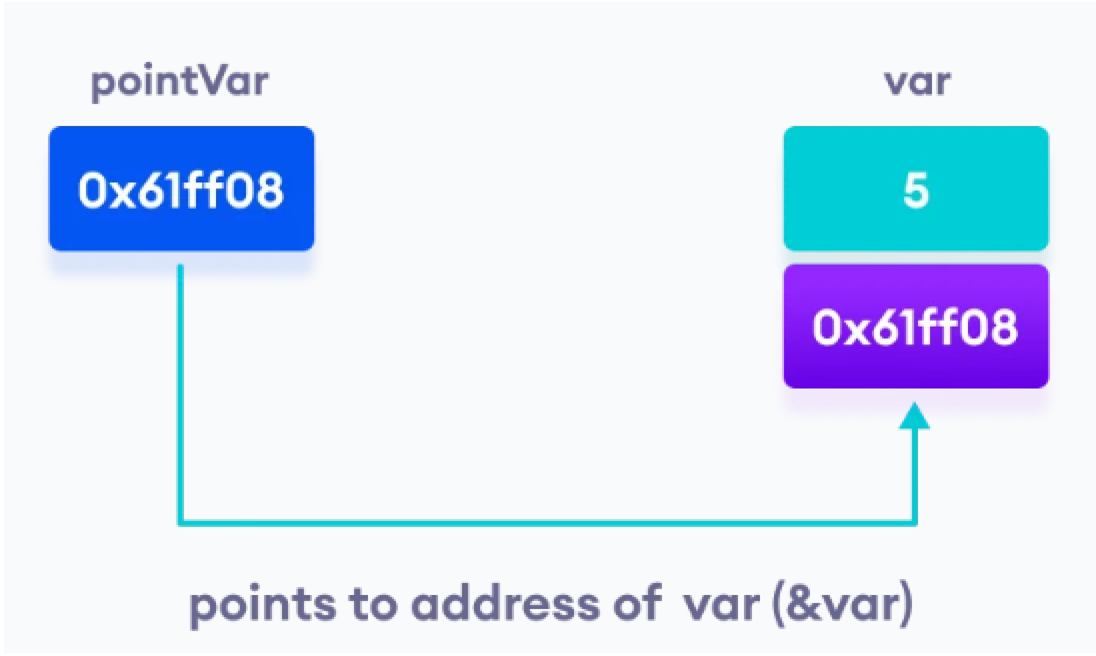


```
int* pointVar; // preferred syntax
```

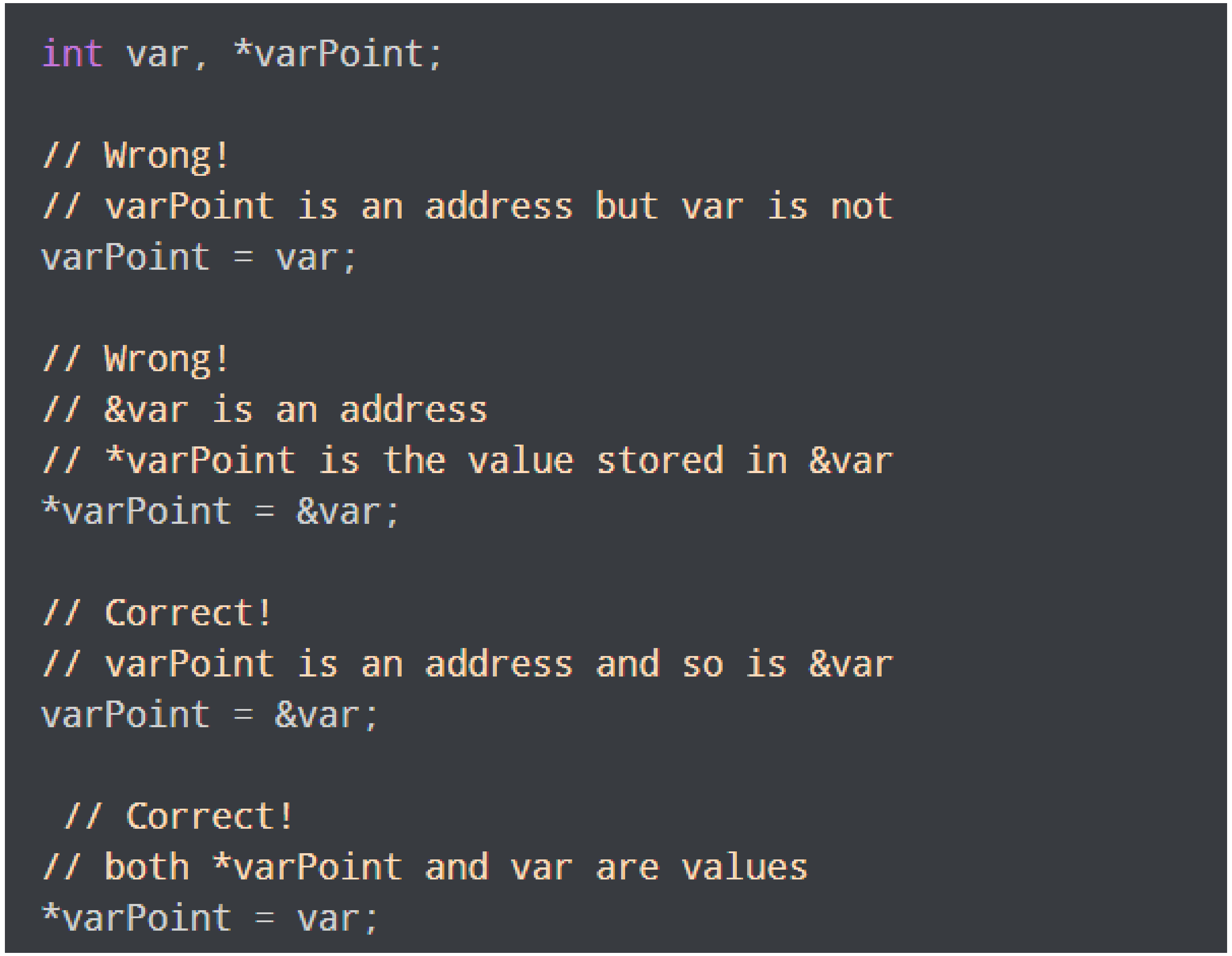
Get the Value from the Address Using Pointers



```
int* pointVar, var;
var = 5;

// assign address of var to pointVar
pointVar = &var;

// access value pointed by pointVar
cout << *pointVar << endl;   // Output: 5
```



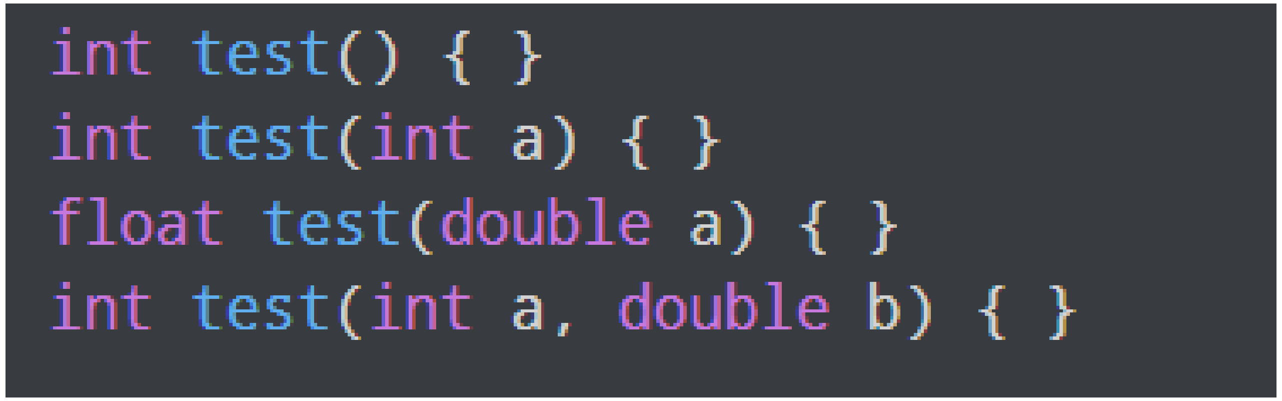
Common mistakes when working with pointers



```c
int var, *varPoint;

// Wrong!
// varPoint is an address but var is not
varPoint = var;

// Wrong!
// &var is an address
// *varPoint is the value stored in &var
*varPoint = &var;

// Correct!
// varPoint is an address and so is &var
varPoint = &var;

 // Correct!
// both *varPoint and var are values
*varPoint = var;
```

Function Overloading

- In C++, two functions can have the same name if the number and/or type of arguments passed is different.
- These functions having the same name but different arguments are known as overloaded functions.

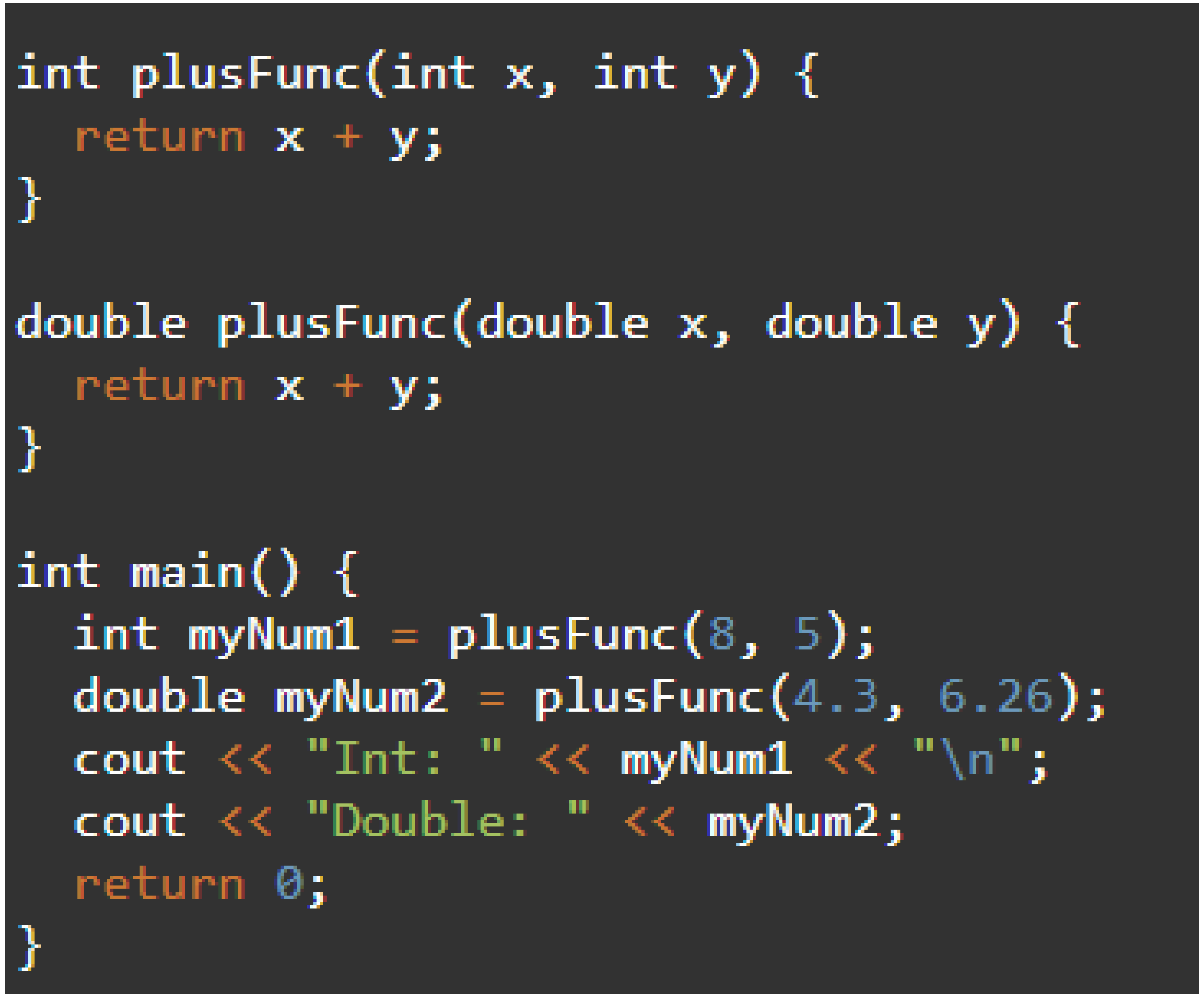


```cpp
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

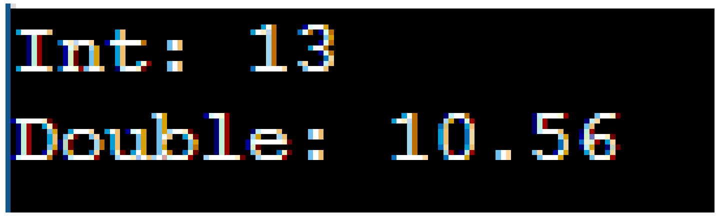
Function Overloading

- Overloaded functions may or may not have different return types but they must have different arguments.

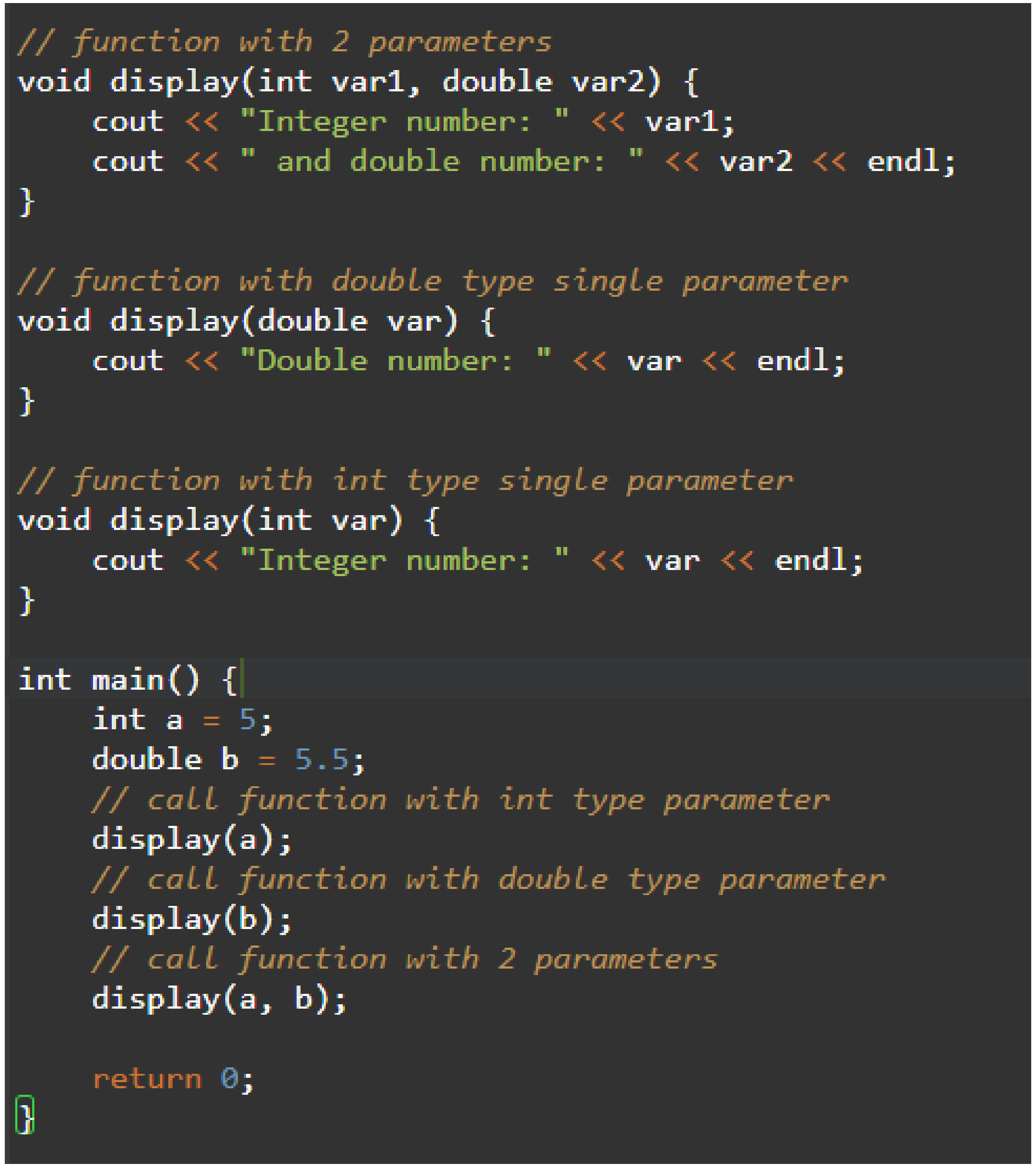
```
// Error code
int test(int a) { }
double test(int b){ }

// correct code
int test(int a) { }
double test(double a){ }
```



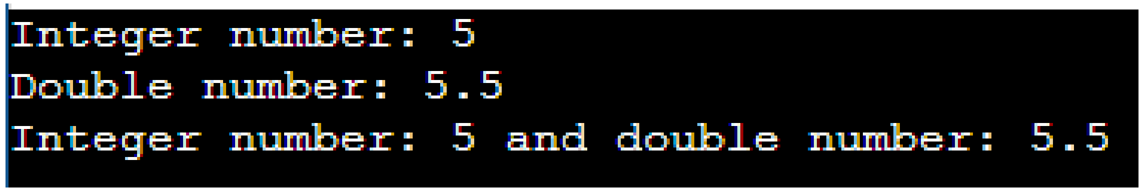
```cpp
int plusFunc(int x, int y) {
  return x + y;
}

double plusFunc(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = plusFunc(8, 5);
  double myNum2 = plusFunc(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```



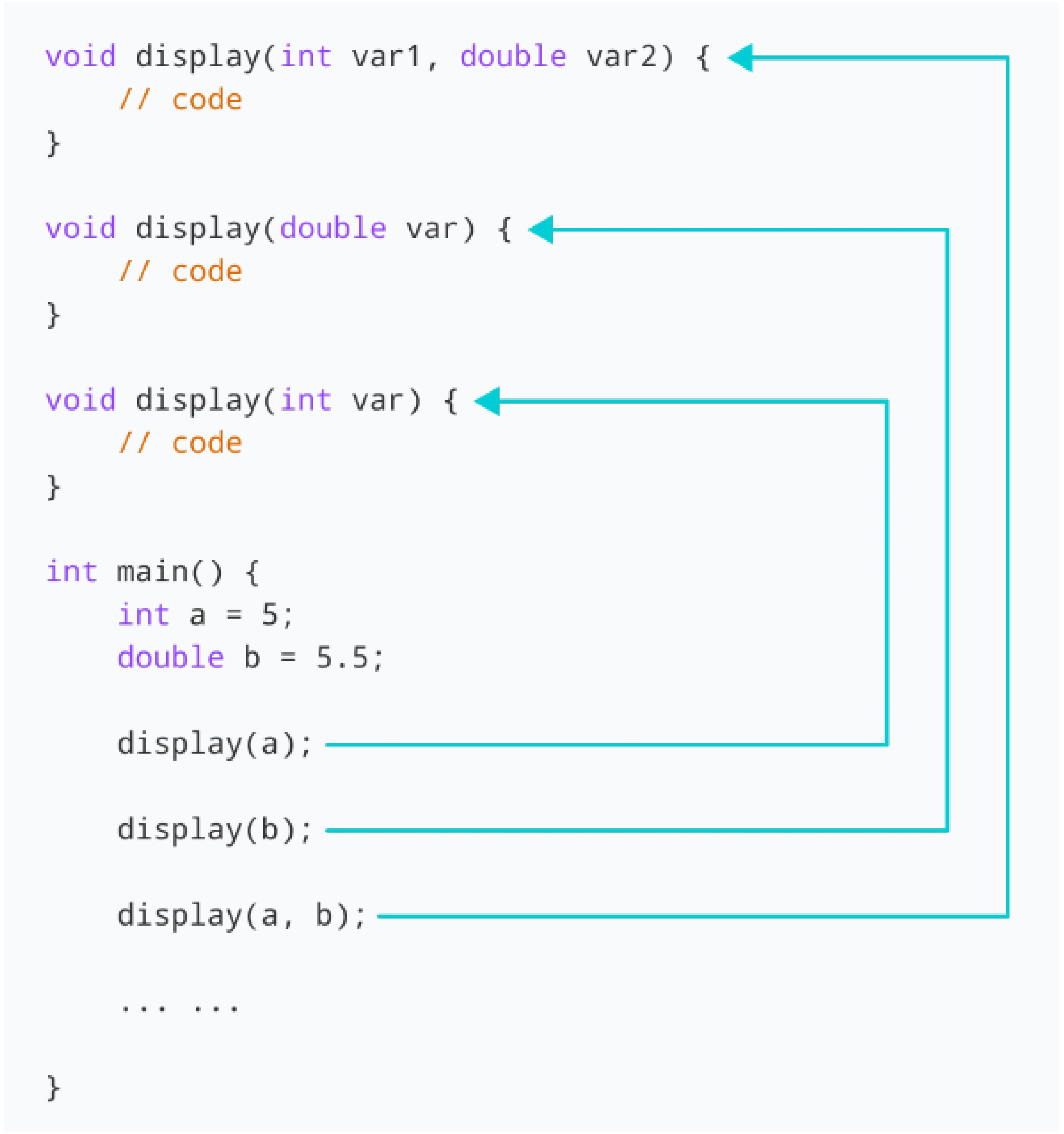
```
Int: 13
Double: 10.56
```



```cpp
// function with 2 parameters
void display(int var1, double var2) {
    cout << "Integer number: " << var1;
    cout << " and double number: " << var2 << endl;
}

// function with double type single parameter
void display(double var) {
    cout << "Double number: " << var << endl;
}

// function with int type single parameter
void display(int var) {
    cout << "Integer number: " << var << endl;
}

int main() {
    int a = 5;
    double b = 5.5;
    // call function with int type parameter
    display(a);
    // call function with double type parameter
    display(b);
    // call function with 2 parameters
    display(a, b);

    return 0;
}
```



```
Integer number: 5
Double number: 5.5
Integer number: 5 and double number: 5.5
```



```
void display(int var1, double var2) {
    // code
}

void display(double var) {
    // code
}

void display(int var) {
    // code
}

int main() {
    int a = 5;
    double b = 5.5;

    display(a);

    display(b);

    display(a, b);

    ... ...

}
```

Namespaces

- Namespaces allow to group entities like classes, objects and functions under a name.
- This way the global scope can be divided in "sub-scopes", each one with its own name.
- Namespaces provide a method for preventing name conflicts in large projects.
- Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes.

Example

- You might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.
- A namespace is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries.
- Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

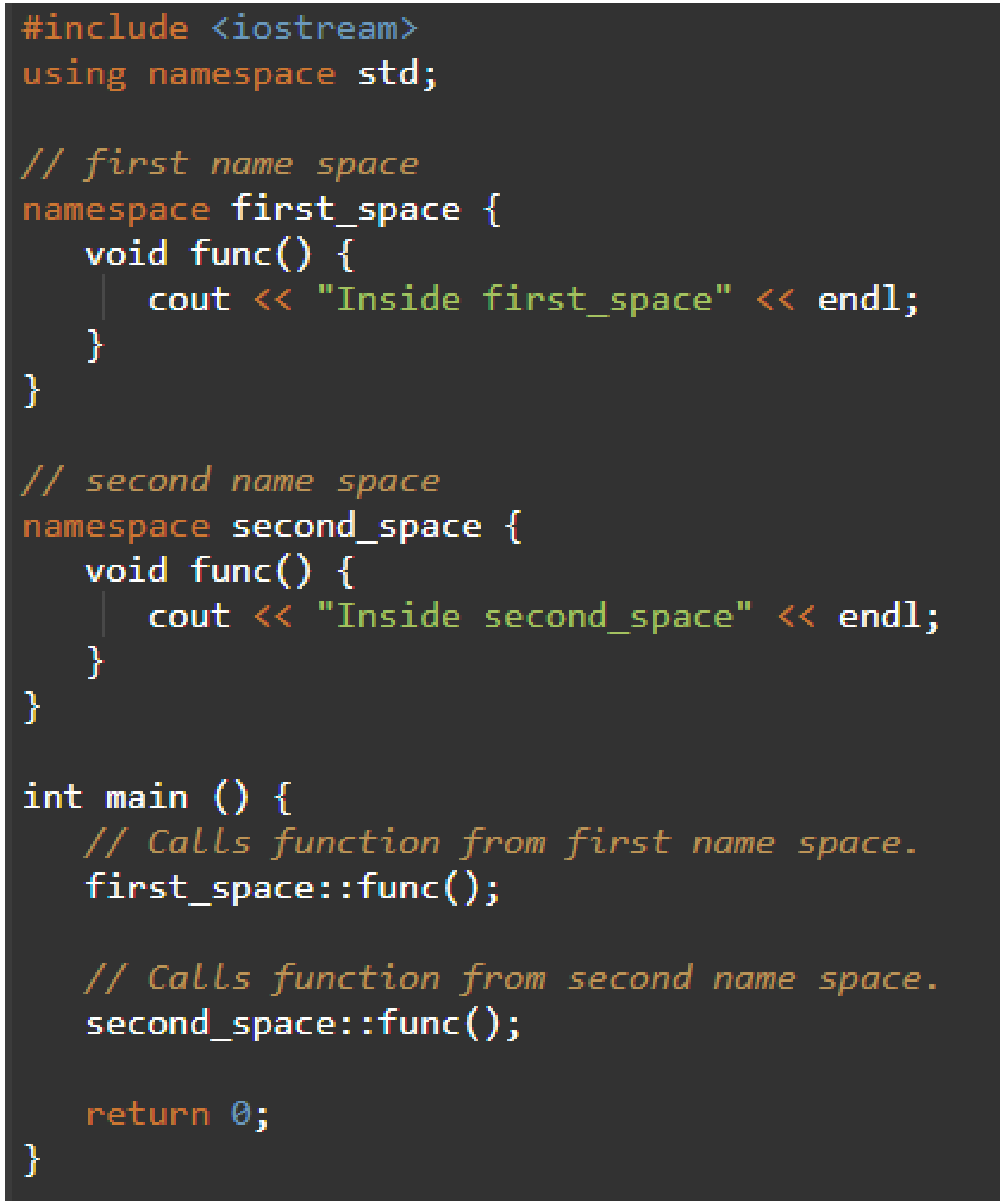
Syntax

A namespace definition begins with the keyword namespace followed by the namespace name:

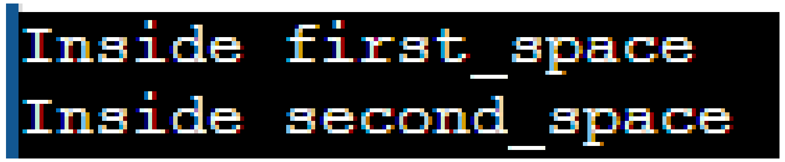
```
namespace namespace_name {
// code declarations
}
```

Calling namespace-enabled function or variable

```
namespace_name::function_name();
```



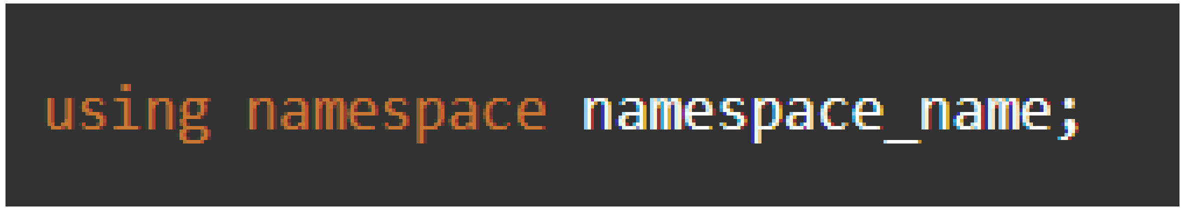
```cpp
#include <iostream>
using namespace std;

// first name space
namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}

// second name space
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}

int main () {
   // Calls function from first name space.
   first_space::func();

   // Calls function from second name space.
   second_space::func();

   return 0;
}
```



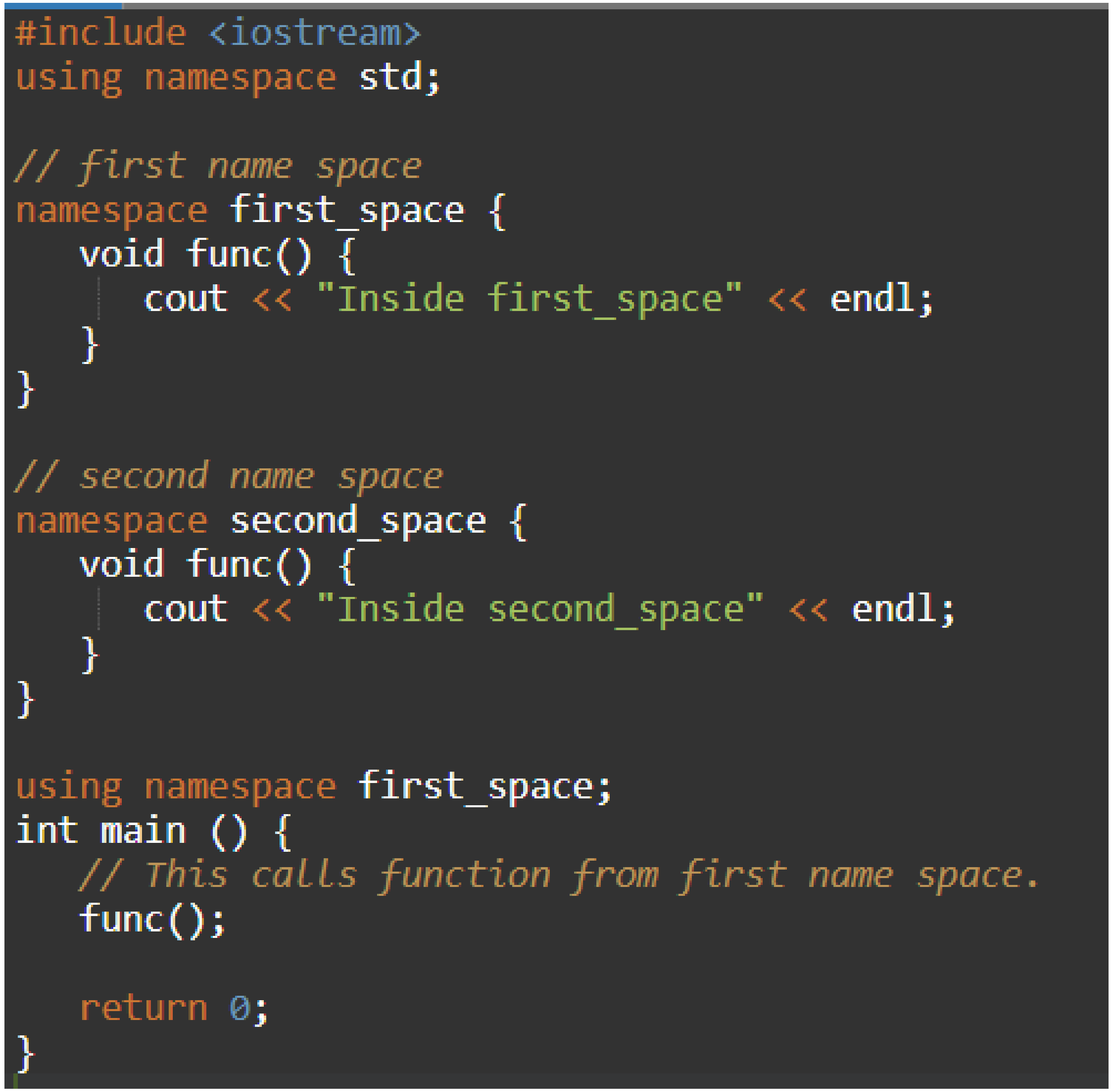
```
Inside first_space
Inside second_space
```

The using directive

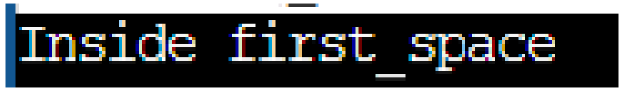
- You can also avoid prepending of namespaces with the using namespace directive.
- This directive tells the compiler that the subsequent code is making use of names in the specified namespace.



```
using namespace namespace_name;
```



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

// first name space
namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}

// second name space
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}

using namespace first_space;
int main () {
   // This calls function from first name space.
   func();

   return 0;
}
```



```
Inside first_space
```

Thanks !!