Functions

The Role of Functions in Code

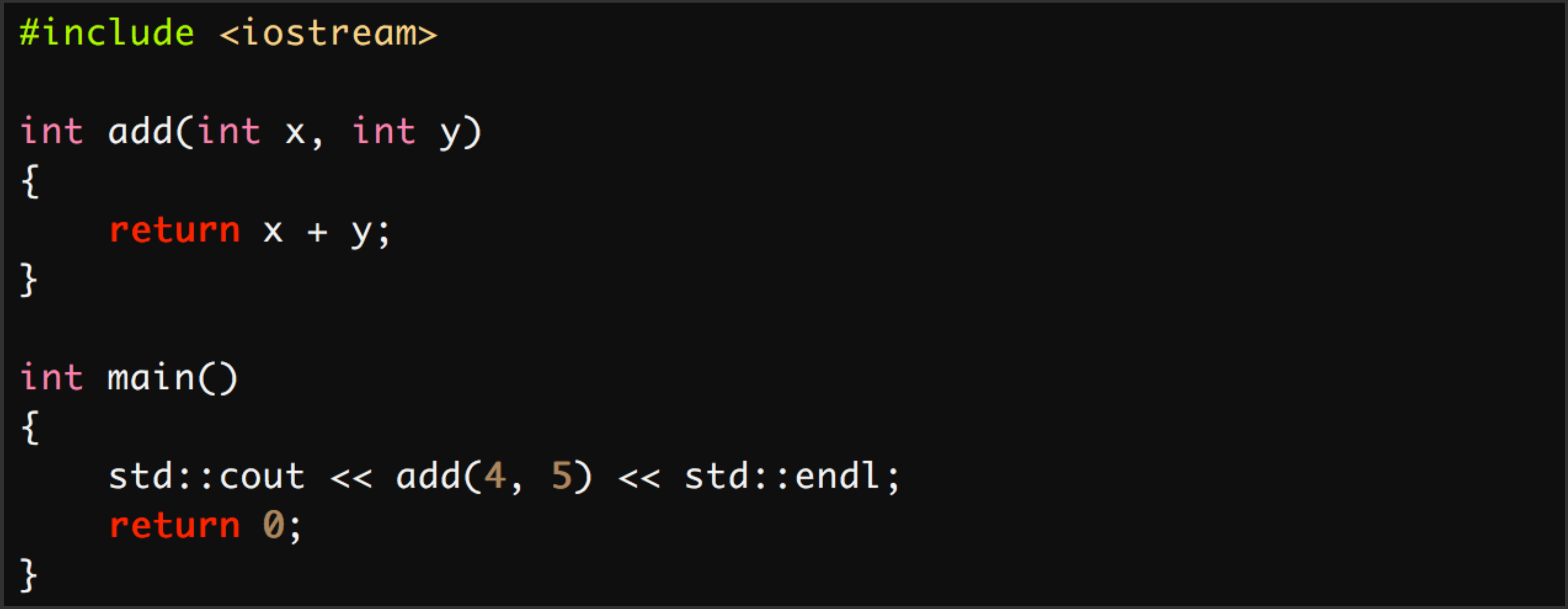
- Functions are used as the central building blocks
- Functions help structure and organize the code

- Functions work as units of data processors
    - they (can) receive input data, process and change that data, and return the processed data or changed input

- **Functions should do one thing and one thing only**
- Functions name should be clear about what the function does

Function Specification

- Function name — int add (int x, int y) { … }
- Return type — int add (int x, int y) { … }
- Parameter (list) — int add (int x, int y) { … }
- Function body — int add (int x, int y) { … }

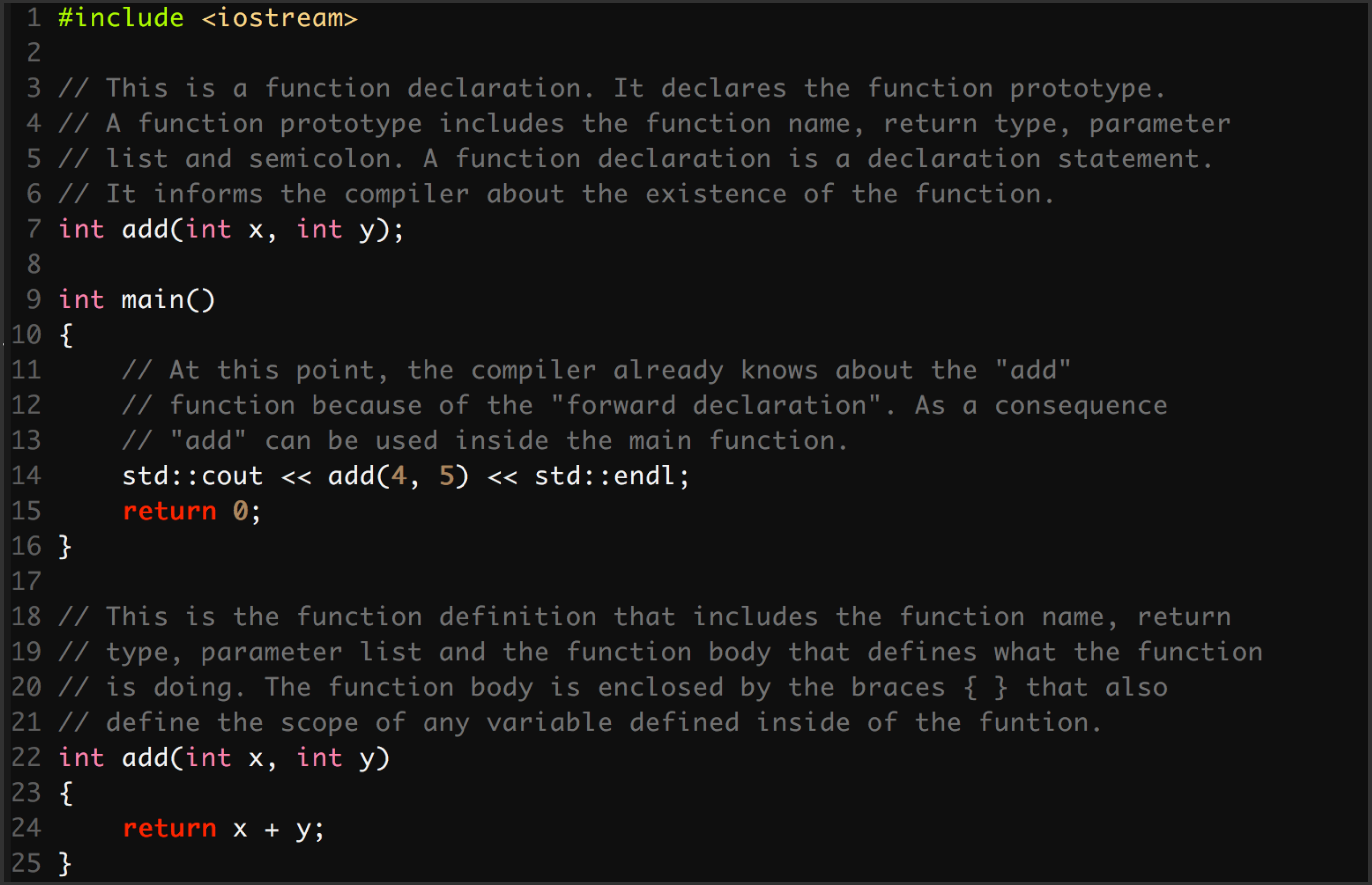


```cpp
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int main()
{
    std::cout << add(4, 5) << std::endl;
    return 0;
}
```

Function Declaration & Definition

- A function declaration informs the compiler about the existence of the function in the code
- It uses a function prototype that includes function name, return type, parameter list & semicolon but no function body
- The function definition includes the function body and can be specified at a later point

Function Declaration & Definition



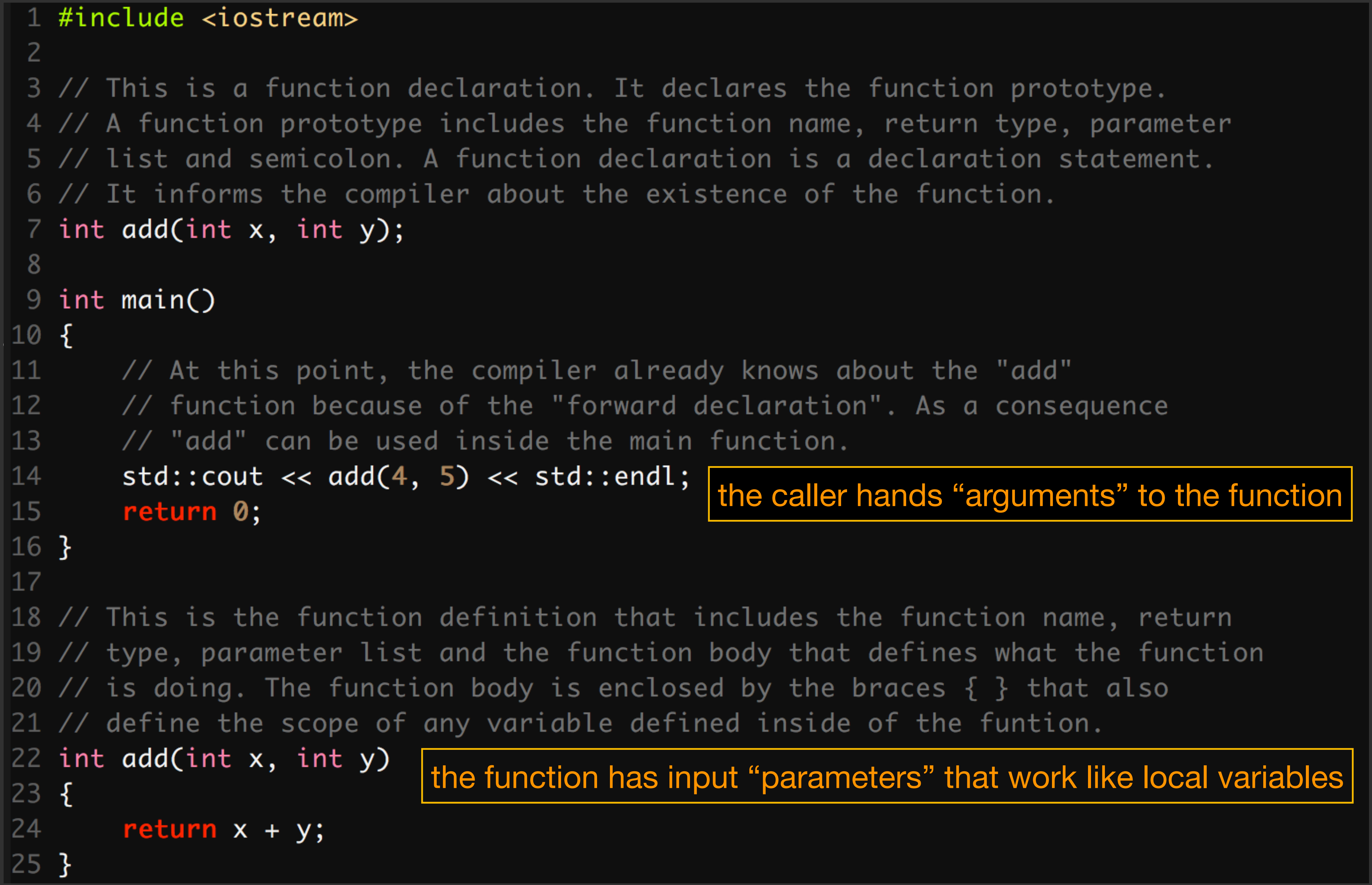
```cpp
#include <iostream>

// This is a function declaration. It declares the function prototype.
// A function prototype includes the function name, return type, parameter
// list and semicolon. A function declaration is a declaration statement.
// It informs the compiler about the existence of the function.
int add(int x, int y);

int main()
{
    // At this point, the compiler already knows about the "add"
    // function because of the "forward declaration". As a consequence
    // "add" can be used inside the main function.
    std::cout << add(4, 5) << std::endl;
    return 0;
}

// This is the function definition that includes the function name, return
// type, parameter list and the function body that defines what the function
// is doing. The function body is enclosed by the braces { } that also
// define the scope of any variable defined inside of the funtion.
int add(int x, int y)
{
    return x + y;
}
```

Function Parameters & Arguments

- Functions can be defined as taking input parameters, i.e., specific variables that will be processed in the function body

- When a function is called that has input parameters, the “caller” hands over or passes certain values called arguments

Function Parameters & Arguments

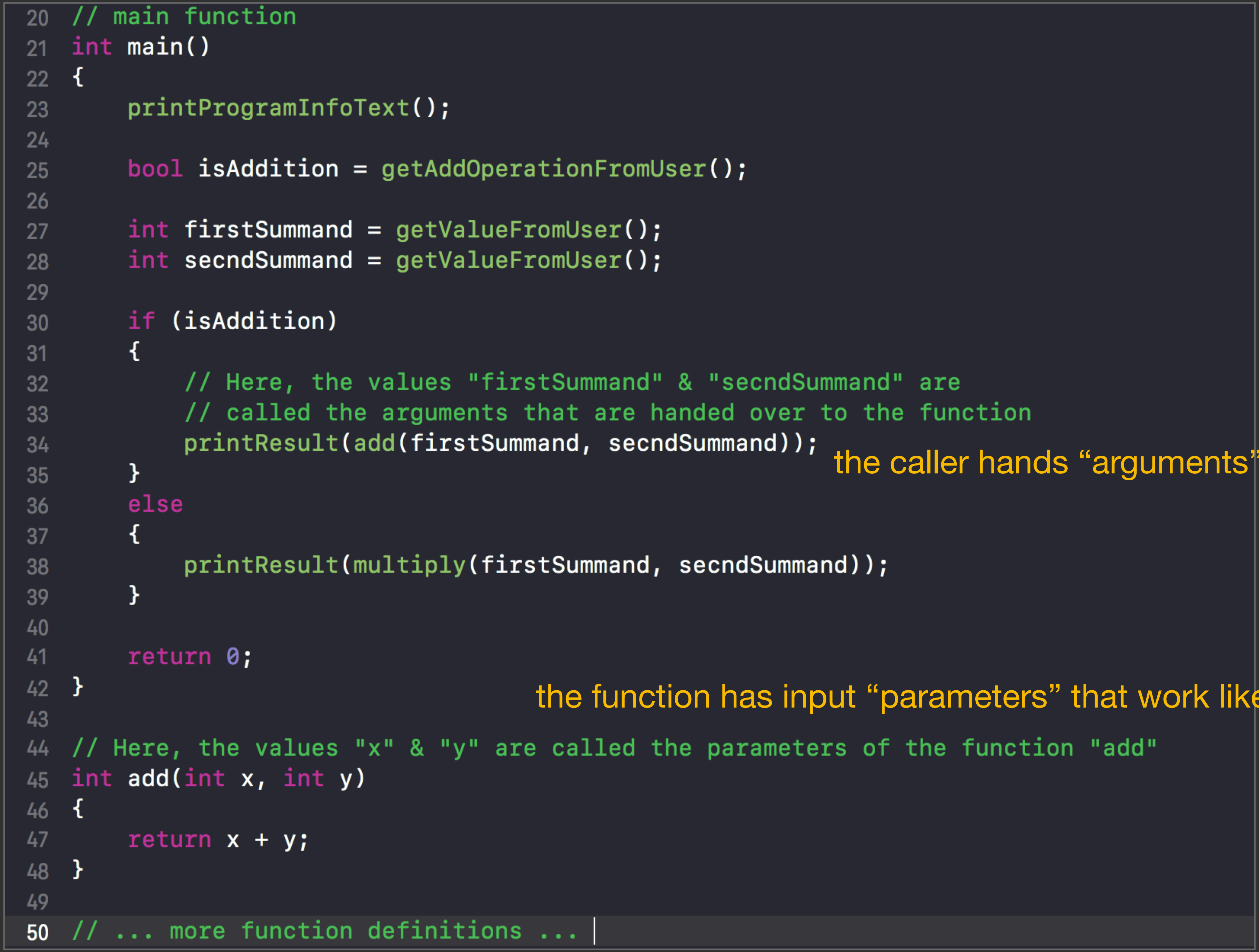


```cpp
#include <iostream>

// This is a function declaration. It declares the function prototype.
// A function prototype includes the function name, return type, parameter
// list and semicolon. A function declaration is a declaration statement.
// It informs the compiler about the existence of the function.
int add(int x, int y);

int main()
{
    // At this point, the compiler already knows about the "add"
    // function because of the "forward declaration". As a consequence
    // "add" can be used inside the main function.
    std::cout << add(4, 5) << std::endl;
    return 0;
}

// This is the function definition that includes the function name, return
// type, parameter list and the function body that defines what the function
// is doing. The function body is enclosed by the braces { } that also
// define the scope of any variable defined inside of the funtion.
int add(int x, int y)
{
    return x + y;
}
```

the caller hands “arguments” to the function

the function has input “parameters” that work like local variables

Function Parameters & Arguments



```
// main function
int main()
{
    printProgramInfoText();

    bool isAddition = getAddOperationFromUser();

    int firstSummand = getValueFromUser();
    int secndSummand = getValueFromUser();

    if (isAddition)
    {
        // Here, the values "firstSummand" & "secndSummand" are
        // called the arguments that are handed over to the function
        printResult(add(firstSummand, secndSummand));
    }
    else
    {
        printResult(multiply(firstSummand, secndSummand));
    }

    return 0;
}

// Here, the values "x" & "y" are called the parameters of the function "add"
int add(int x, int y)
{
    return x + y;
}

// ... more function definitions ...
```

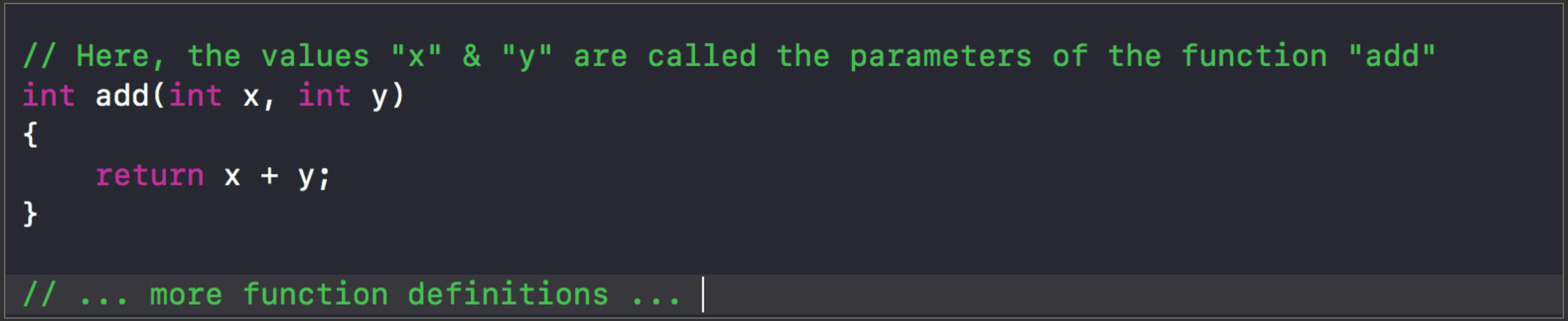
the caller hands “arguments” to the function

the function has input “parameters” that work like local variables

Function Parameters & Arguments

- There are two basic forms of calling a function
  - **Pass-by-value**
  - **Pass-by-reference**

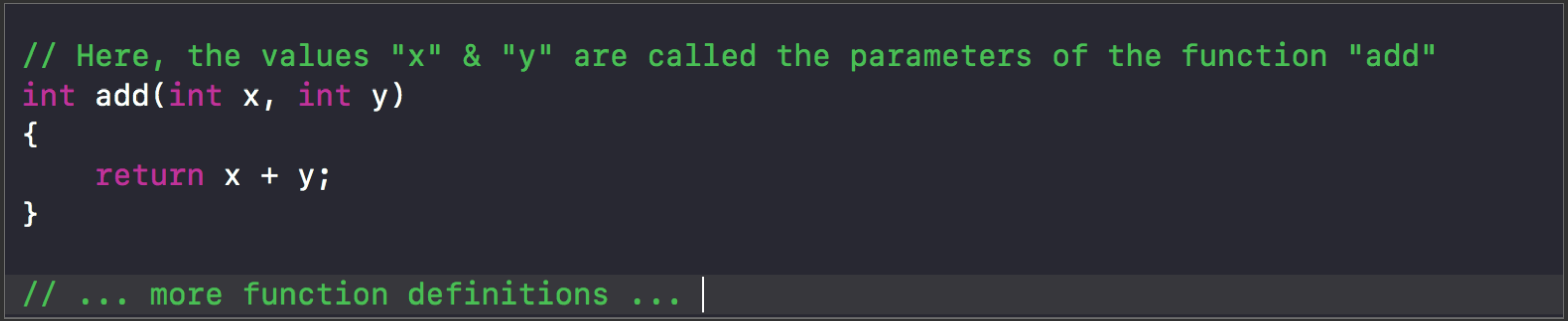
Pass-by-Value



```
// Here, the values "x" & "y" are called the parameters of the function "add"
int add(int x, int y)
{
    return x + y;
}

// ... more function definitions ...
```

- Pass-by-value means that the function creates a **local copy** of the values passed to it
  - The parameters “x” & “y” are only accessible inside of “add”
  - The arguments in the calling function are never changed

Pass-by-Value

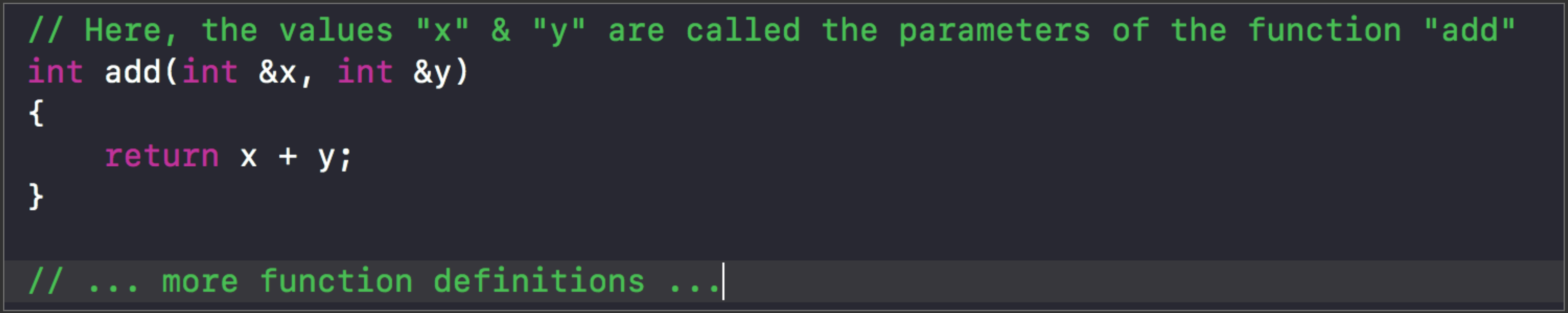


```
// Here, the values "x" & "y" are called the parameters of the function "add"
int add(int x, int y)
{
    return x + y;
}

// ... more function definitions ...
```

- When to use it?
  - When small data types are passed (fundamental data types)
  - When the function does not need to change the arguments

- When not to use it?
  - When passing large data types (i.e., arrays, structs, objects,…)

Pass-by-Reference

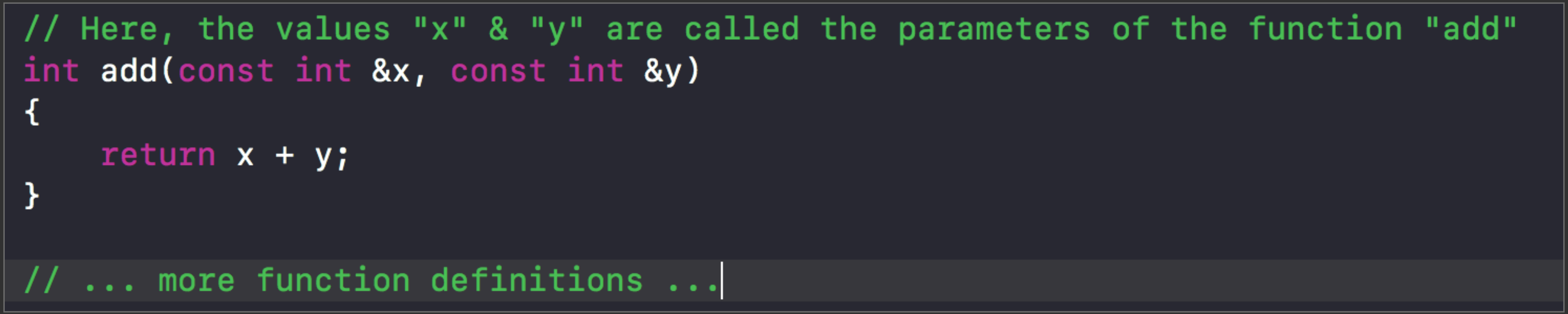


```
// Here, the values "x" & "y" are called the parameters of the function "add"
int add(int &x, int &y)
{
    return x + y;
}

// ... more function definitions ...
```

- Pass-by-reference means that the parameters become **reference variables** that directly reference the arguments

- Pass-by-reference introduces the reference operator &
- The function „add“ will now work on the original values of the calling function; any changes are reflected back to the caller

Pass-by-Reference

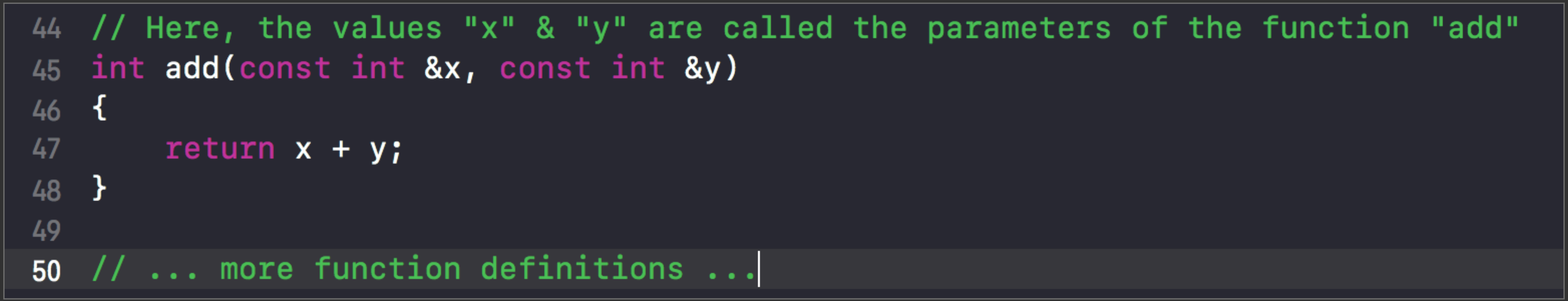


```
// Here, the values "x" & "y" are called the parameters of the function "add"
int add(const int &x, const int &y)
{
    return x + y;
}

// ... more function definitions ...
```

- Reference variables can be turned into **constants** using the const keyword to avoid that they can be changed

- In combination with functions this is used to
  - to avoid a pass-by-value — copy — operation
  - to avoid any changes to the arguments

Pass-by-Reference

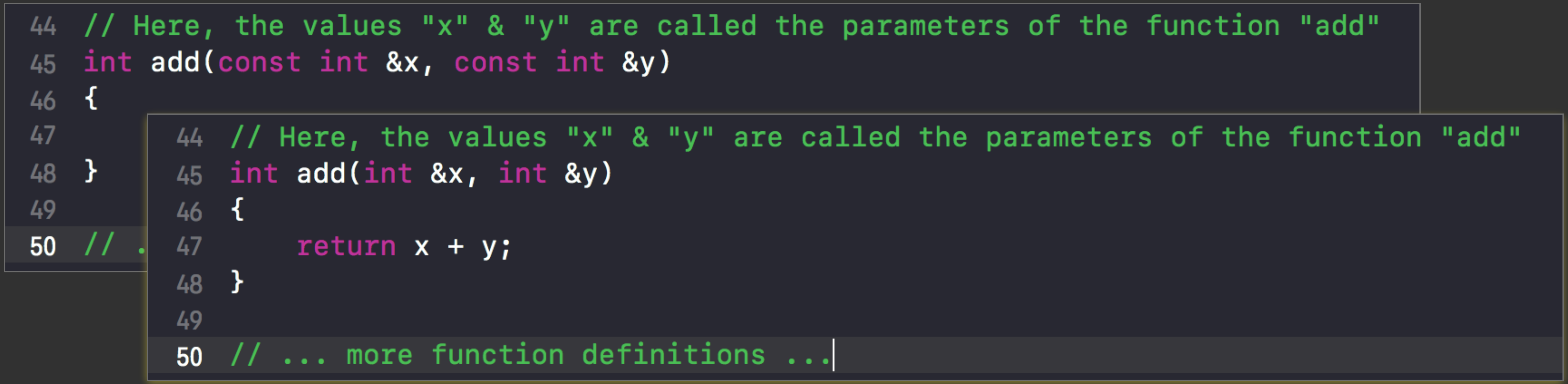


```
// Here, the values "x" & "y" are called the parameters of the function "add"
int add(const int &x, const int &y)
{
    return x + y;
}

// ... more function definitions ...
```

- When to use it?
  - With large data types usually in combination with “const”
  - When the function should not change the arguments
  - When more than one return value is required

    (This is usually **bad design** and should be avoided if possible)

Pass-by-Reference



```cpp
// Here, the values "x" & "y" are called the parameters of the function "add"
int add(const int &x, const int &y)
{

}

// .
```

```cpp
// Here, the values "x" & "y" are called the parameters of the function "add"
int add(int &x, int &y)
{
    return x + y;
}

// ... more function definitions ...
```

- When not to use it (or when you should think twice)?
  - When you want to use more than one return value
  - When you are working with fundamental data types