Function Overloading

It’s not dangerous, I promise…

Function Overloading:

Making more than one function *with a given name*

Function Overloading

You can have multiple functions with the same name!

- this is called “function overloading”

To differentiate between functions, C++ uses:

- the name of the function (pretty obvious ^_^)
- the number of arguments
- the data types of arguments
- the const-ness of pass-by-reference arguments

C++ does not use:

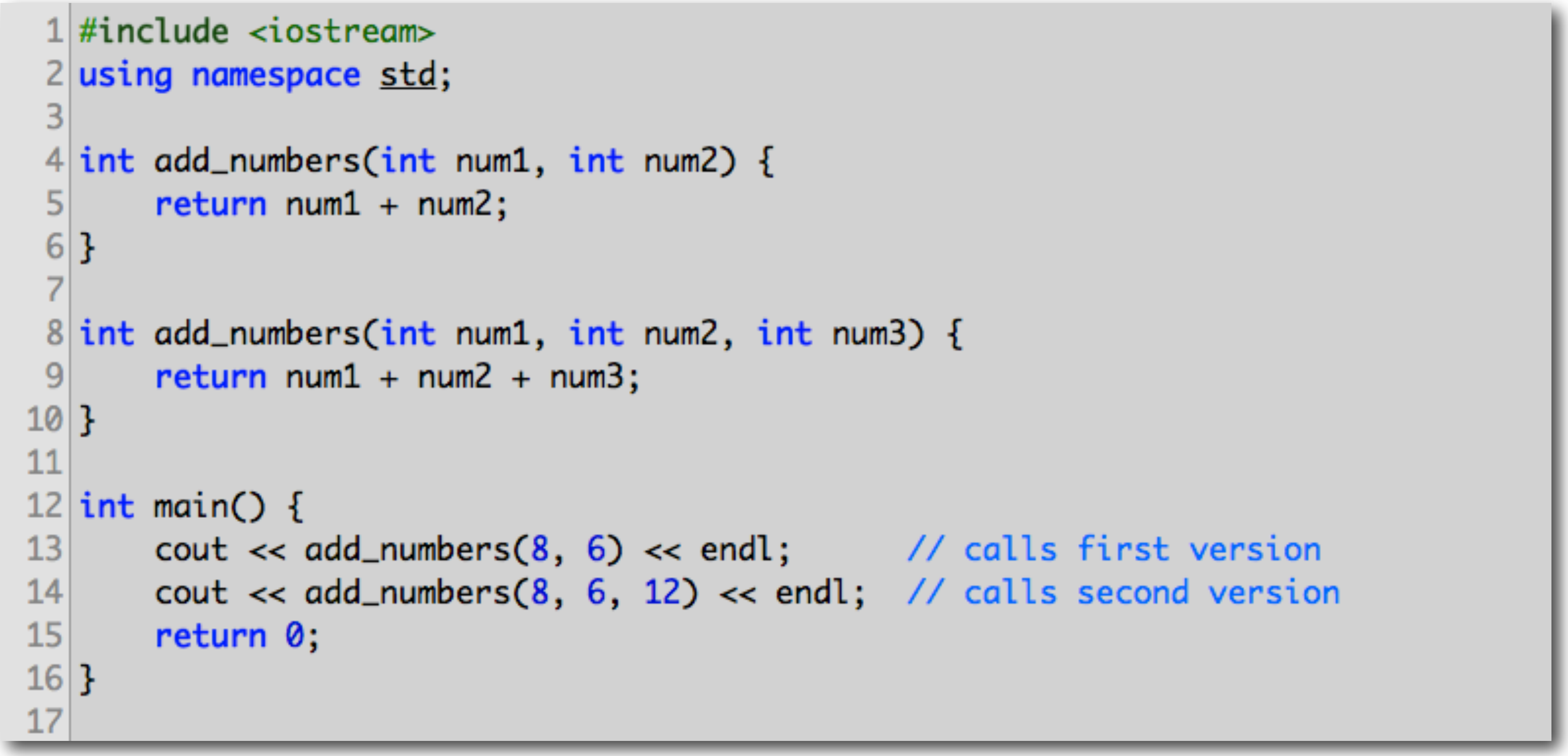
- the return type (including const-ness and return-by-reference)
- argument names (names are only for our benefit)
- const-ness of pass-by-value arguments

Function Overloading

These are unique functions (different *number* of arguments):

```
int add_numbers(int num1, int num2);

int add_numbers(int num1, int num2, int num3);
```

Example:



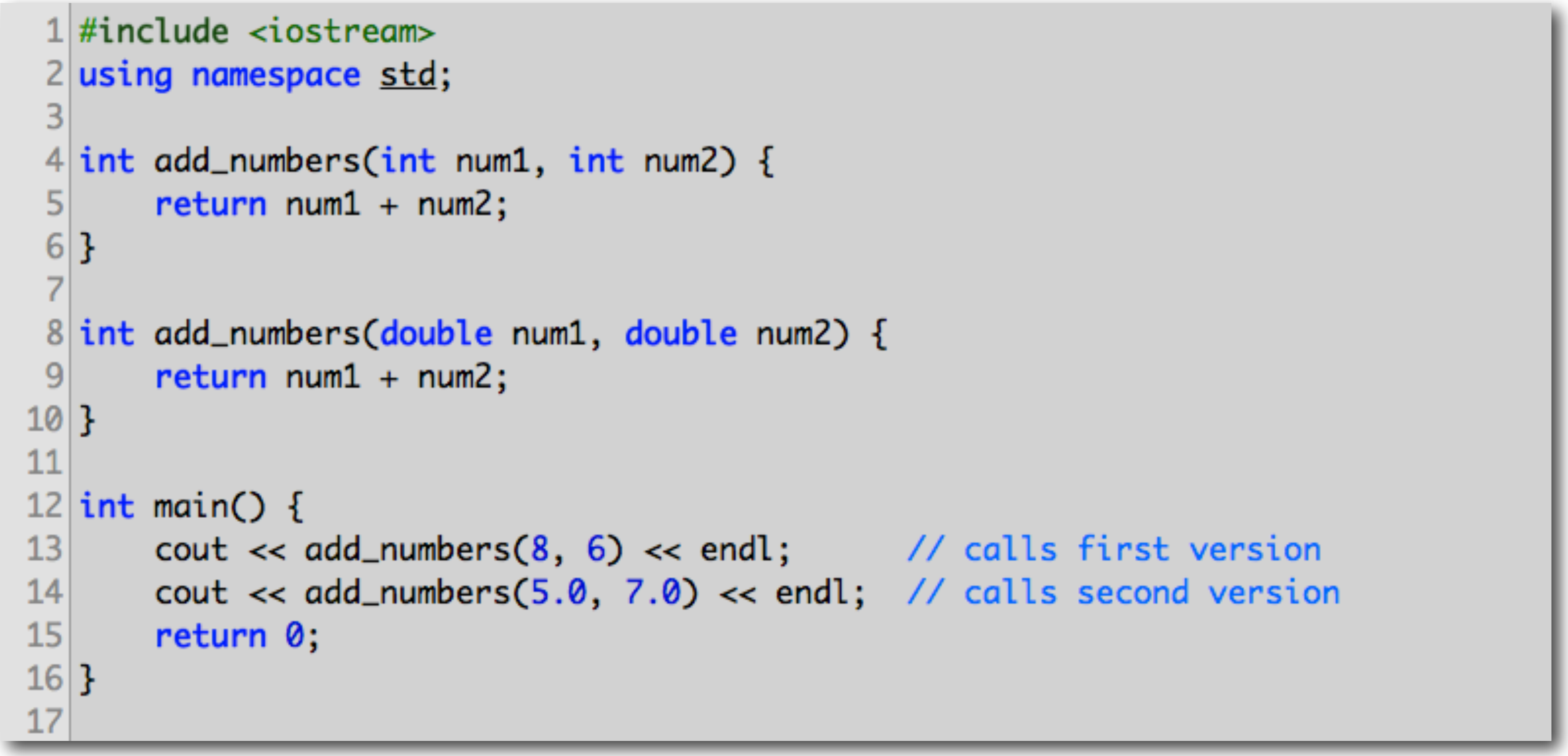
```
#include <iostream>
using namespace std;

int add_numbers(int num1, int num2) {
    return num1 + num2;
}

int add_numbers(int num1, int num2, int num3) {
    return num1 + num2 + num3;
}

int main() {
    cout << add_numbers(8, 6) << endl;     // calls first version
    cout << add_numbers(8, 6, 12) << endl; // calls second version
    return 0;
}

```

Function Overloading

These are unique functions (different *types* of arguments):

```
int add_numbers(int num1, int num2);

int add_numbers(double num1, double num2);
```

Example:



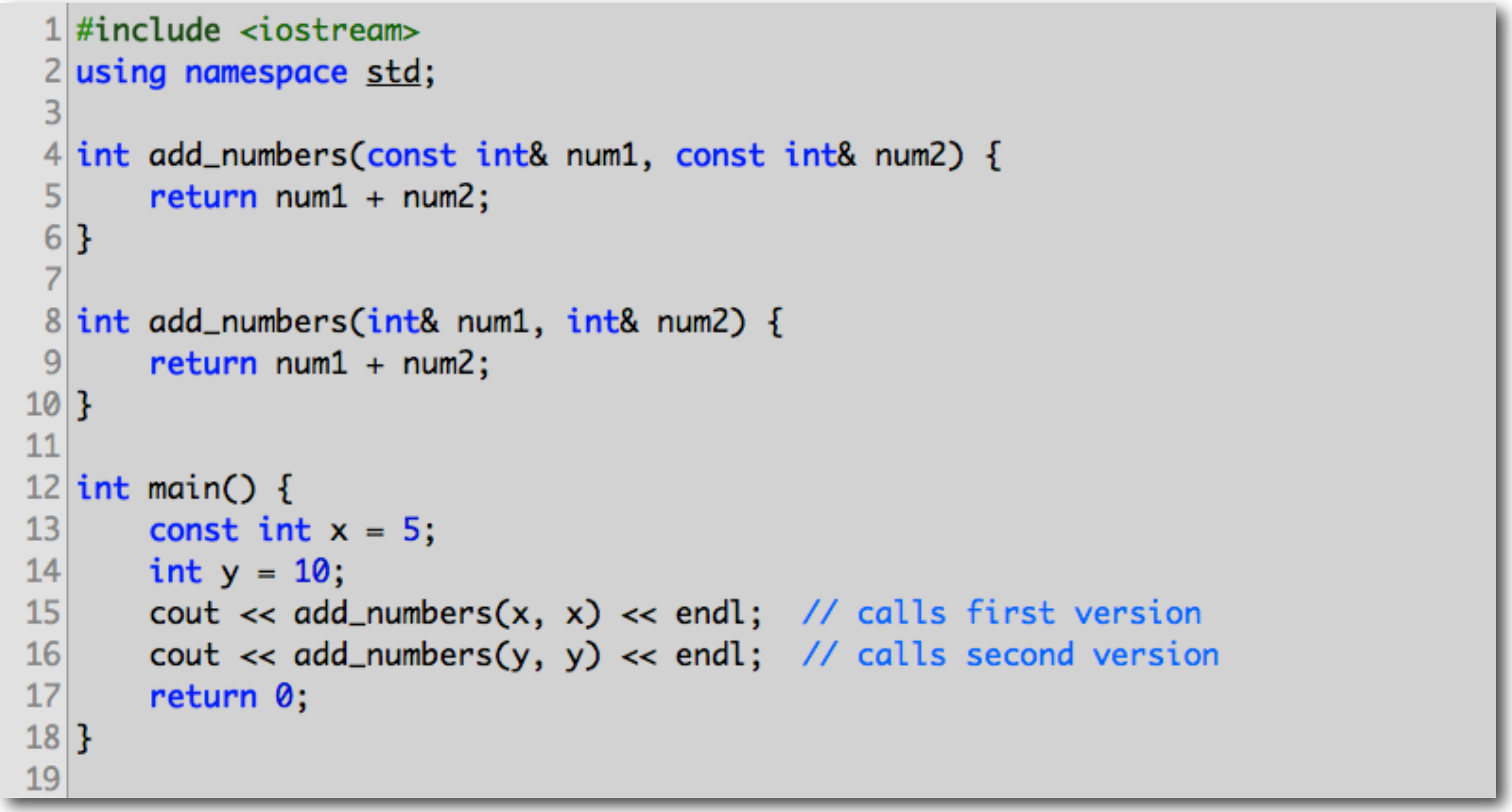
```
#include <iostream>
using namespace std;

int add_numbers(int num1, int num2) {
    return num1 + num2;
}

int add_numbers(double num1, double num2) {
    return num1 + num2;
}

int main() {
    cout << add_numbers(8, 6) << endl;      // calls first version
    cout << add_numbers(5.0, 7.0) << endl;  // calls second version
    return 0;
}

```

Function Overloading

These are unique functions (different *const-ness* of &arguments):

```
int add_numbers(const int& num1, const int& num2);

int add_numbers(int& num1, int& num2);
```

Example:



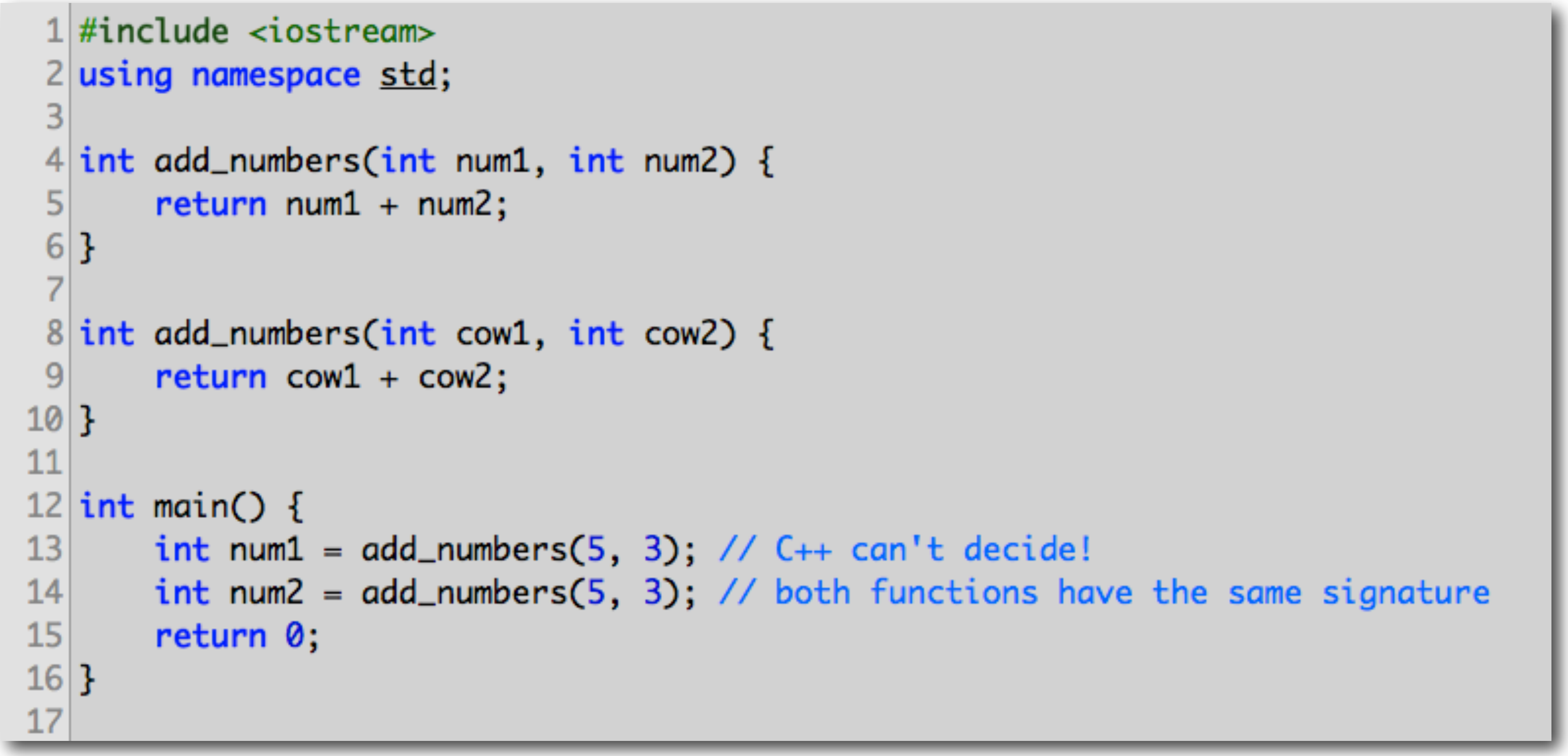
```
#include <iostream>
using namespace std;

int add_numbers(const int& num1, const int& num2) {
    return num1 + num2;
}

int add_numbers(int& num1, int& num2) {
    return num1 + num2;
}

int main() {
    const int x = 5;
    int y = 10;
    cout << add_numbers(x, x) << endl;  // calls first version
    cout << add_numbers(y, y) << endl;  // calls second version
    return 0;
}

```

Not Function Overloading

These are NOT unique functions (*argument names* don’t matter):

```
int add_numbers(int num1, int num2);

int add_numbers(int cow1, int cow2);
```

Example:



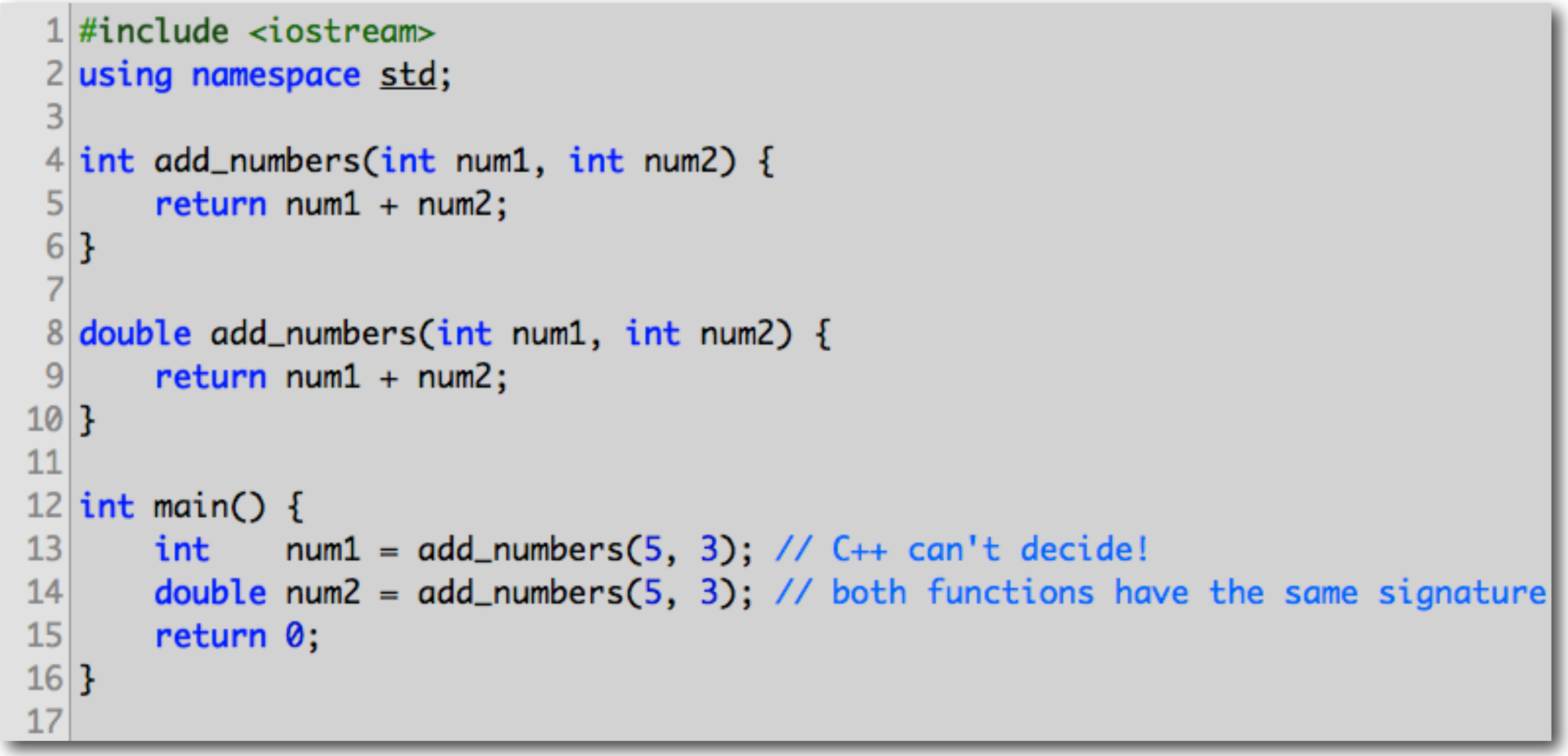
```
#include <iostream>
using namespace std;

int add_numbers(int num1, int num2) {
    return num1 + num2;
}

int add_numbers(int cow1, int cow2) {
    return cow1 + cow2;
}

int main() {
    int num1 = add_numbers(5, 3); // C++ can't decide!
    int num2 = add_numbers(5, 3); // both functions have the same signature
    return 0;
}
```

Not Function Overloading

These are NOT unique functions (*return type* doesn’t matter):

```
int    add_numbers(int num1, int num2);
double add_numbers(int num1, int num2);
```

Example:



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

int add_numbers(int num1, int num2) {
    return num1 + num2;
}

double add_numbers(int num1, int num2) {
    return num1 + num2;
}

int main() {
    int    num1 = add_numbers(5, 3); // C++ can't decide!
    double num2 = add_numbers(5, 3); // both functions have the same signature
    return 0;
}

```

Overloading acos()

Let’s say I wanted to be really REALLY lazy in the future…

So I overload cmath’s acos() to return PI if no argument is passed:

```
double acos() {
    return acos(-1); // calls the other version of acos
}
```

Now I can get PI in two different ways:

```
const double PI  = acos(-1); // like we’ve done since day 1
const double PIE = acos();   // uses the overloaded acos() fn!
```

Yay for pie! =)