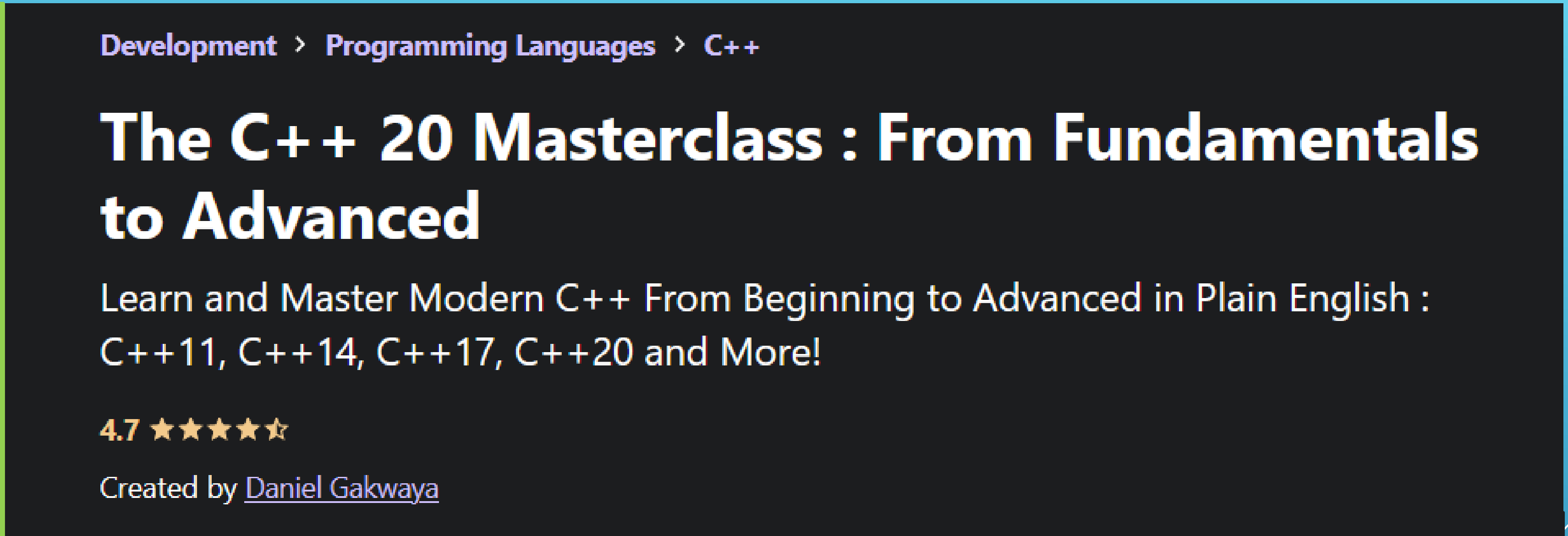
Slides



Development › Programming Languages › C++

The C++ 20 Masterclass : From Fundamentals to Advanced

Learn and Master Modern C++ From Beginning to Advanced in Plain English : C++11, C++14, C++17, C++20 and More!

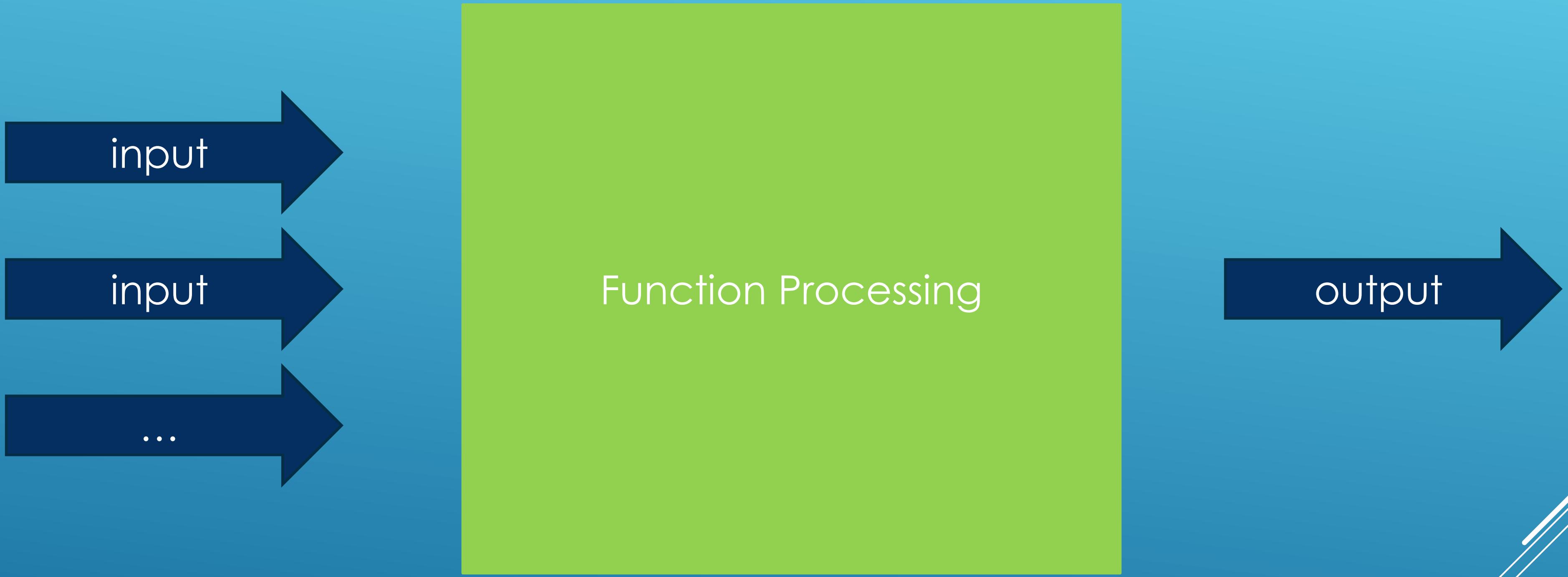
4.7 ★★★★☆

Created by Daniel Gakwaya

Section : Functions

Slide intentionally left empty

Functions : Introduction


input
input
...
Function Processing
output

Top down programming in the main function

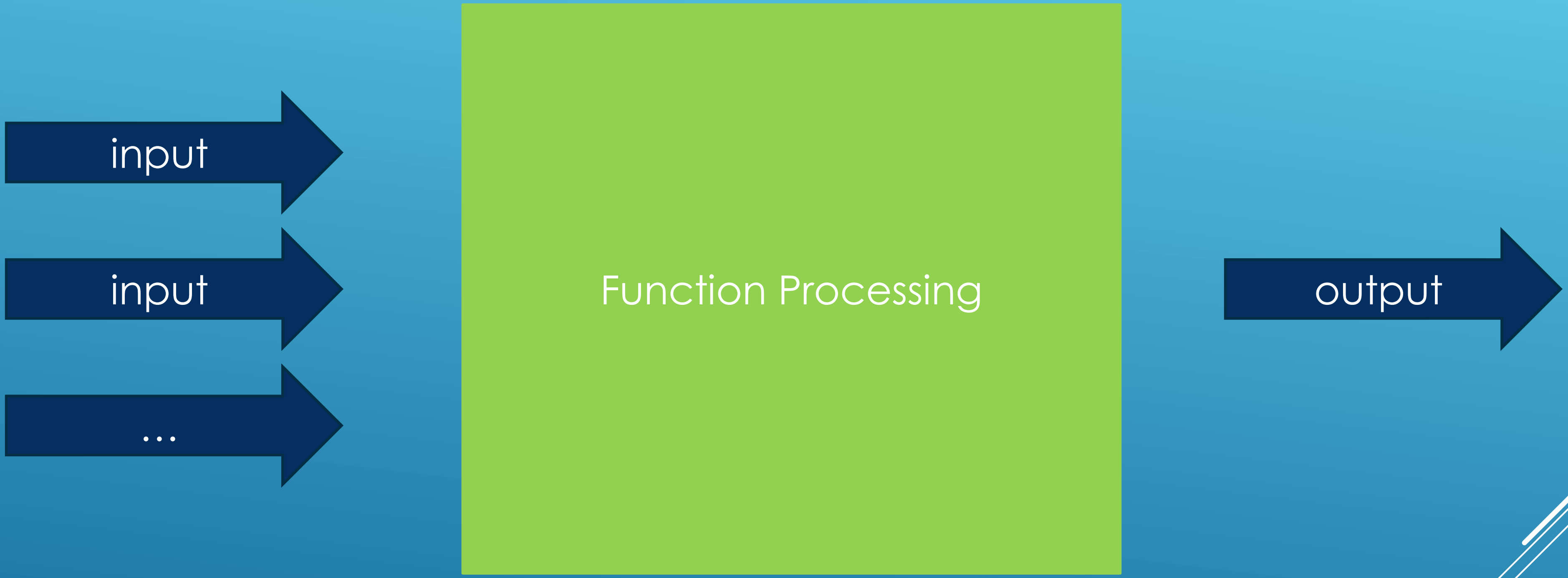
Reusable code components we can call several times in the main function

Slide intentionally left empty

First Hand on Functions

Function

A reusable piece of code that can take a number of optional inputs and produce some desirable output



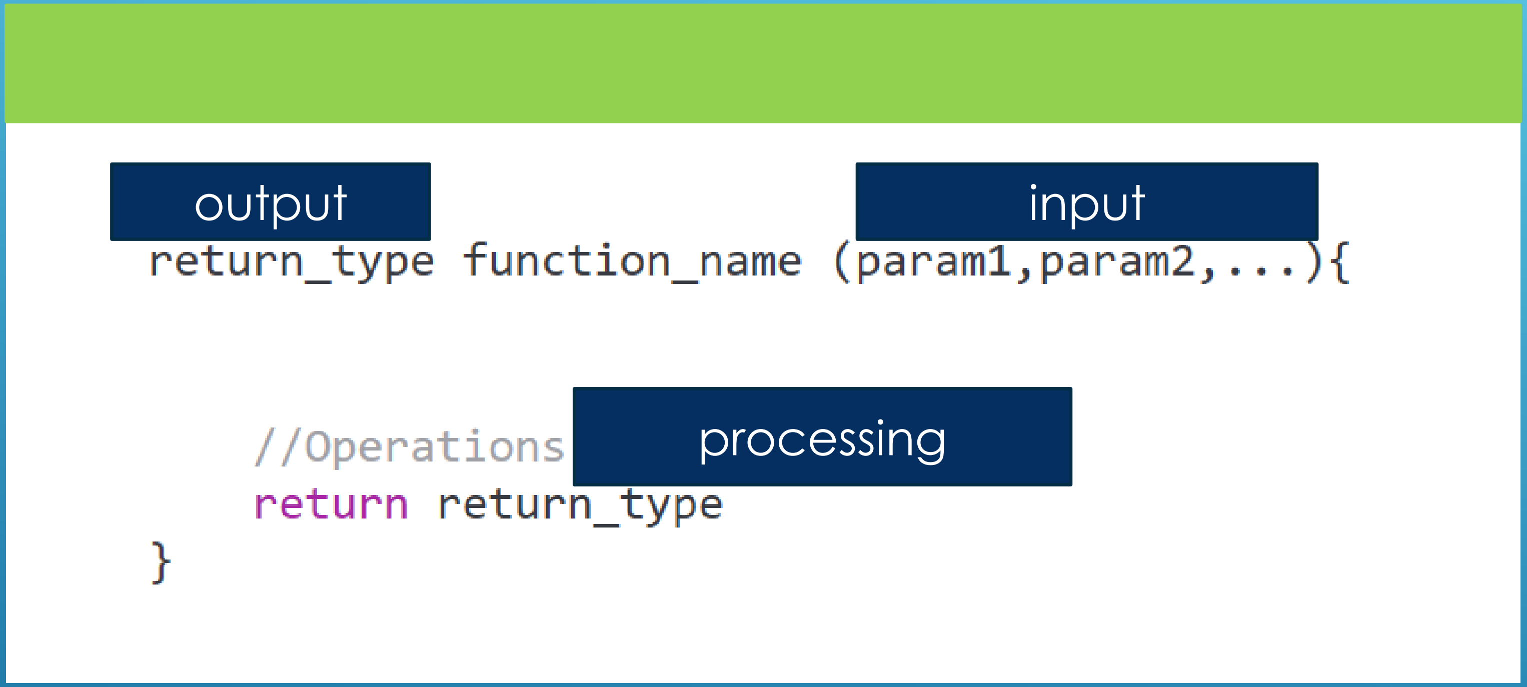
input

input

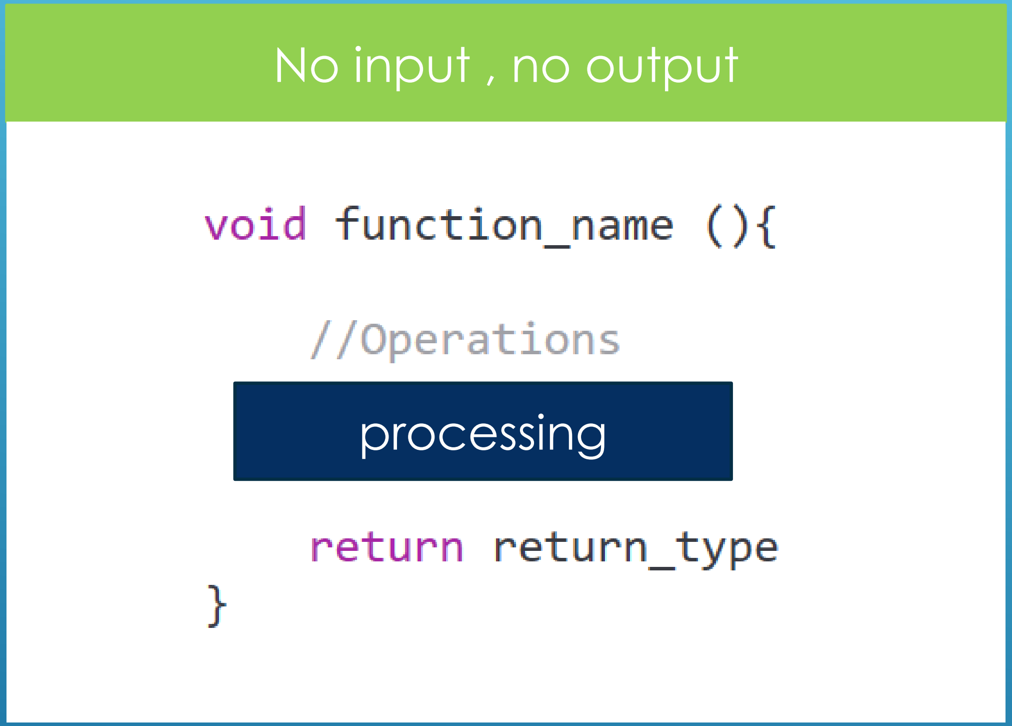
...

Function Processing

output



```
return_type function_name (param1,param2,...){


    //Operations
    return return_type
}
```



No input , no output

```
void function_name (){

    //Operations
```

processing

```
    return return_type
}
```

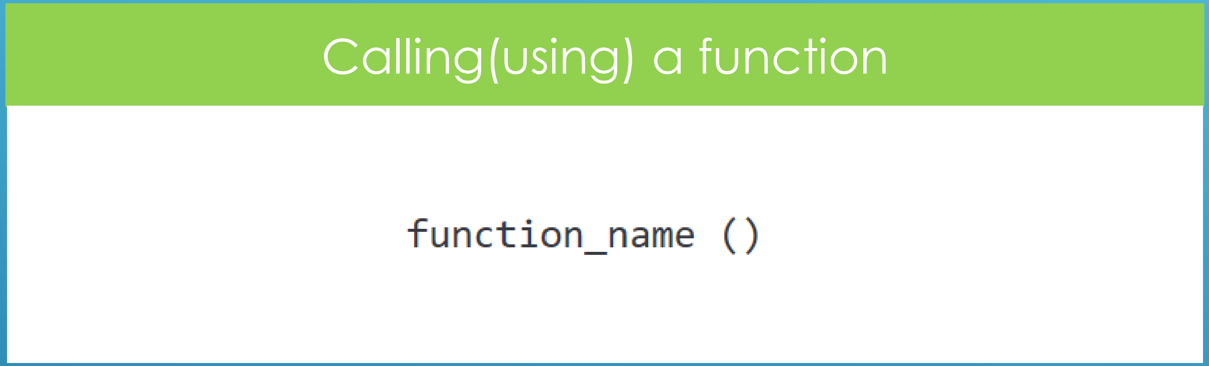
Function signature	Function name + function parameters

Function parameters

When you are setting up your function (declaration), the input units are called parameters. A legal C++ function can have 0 or more parameters

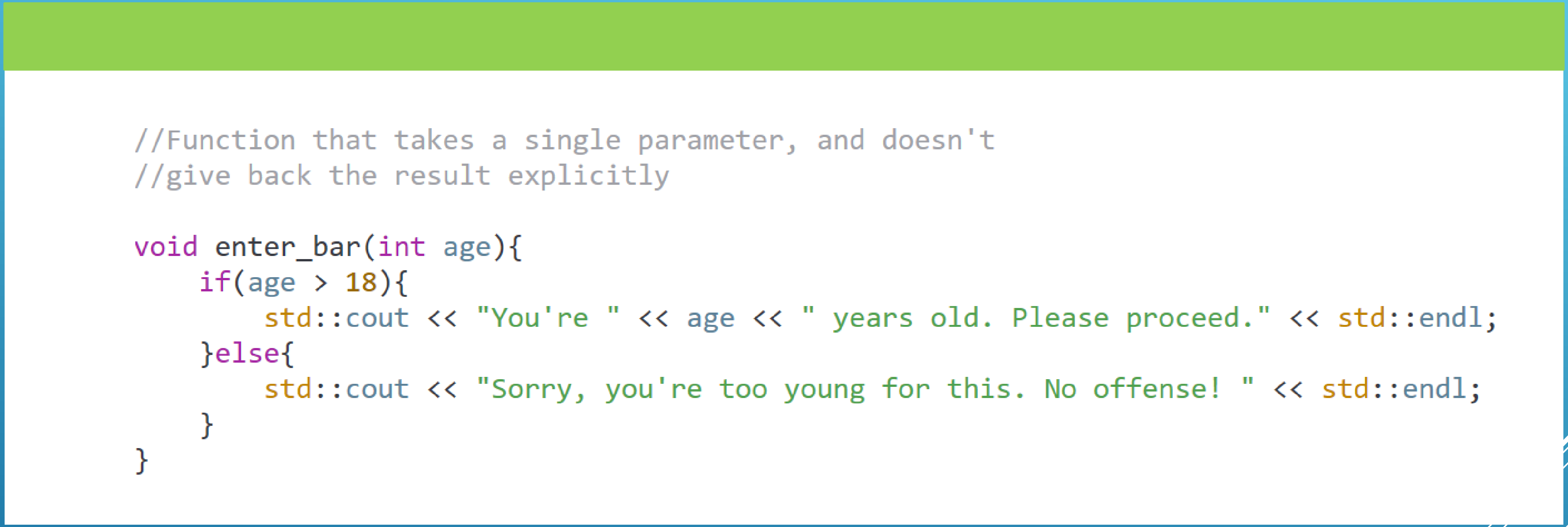
Calling(using) a function

```
result_var = function_name (arg1,arg2)
```

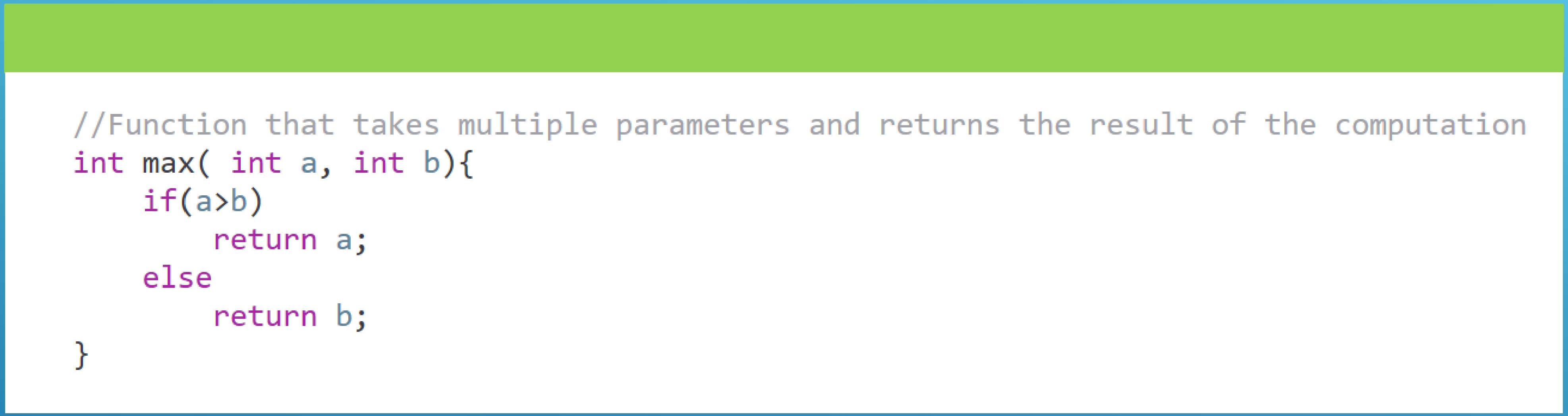


Calling(using) a function

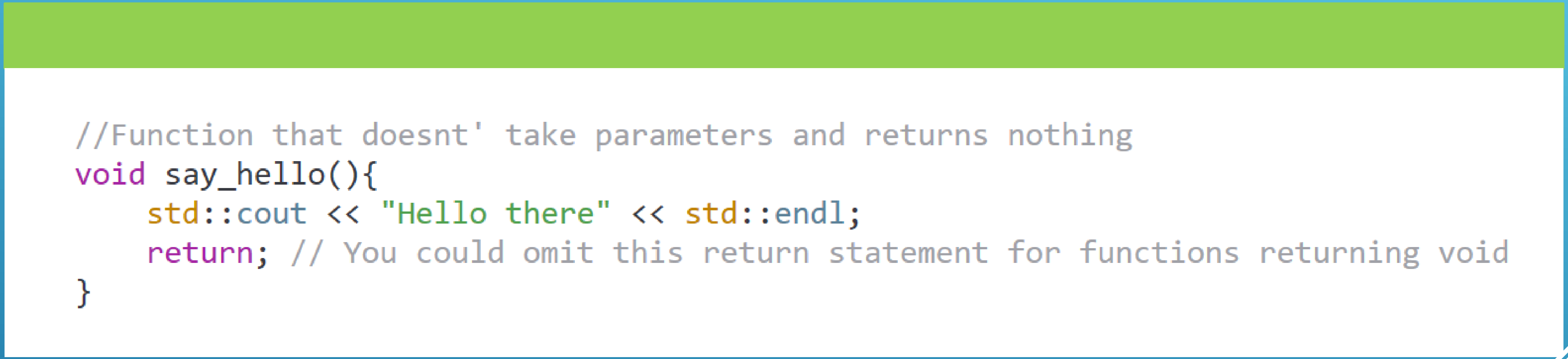
```
function_name ()
```



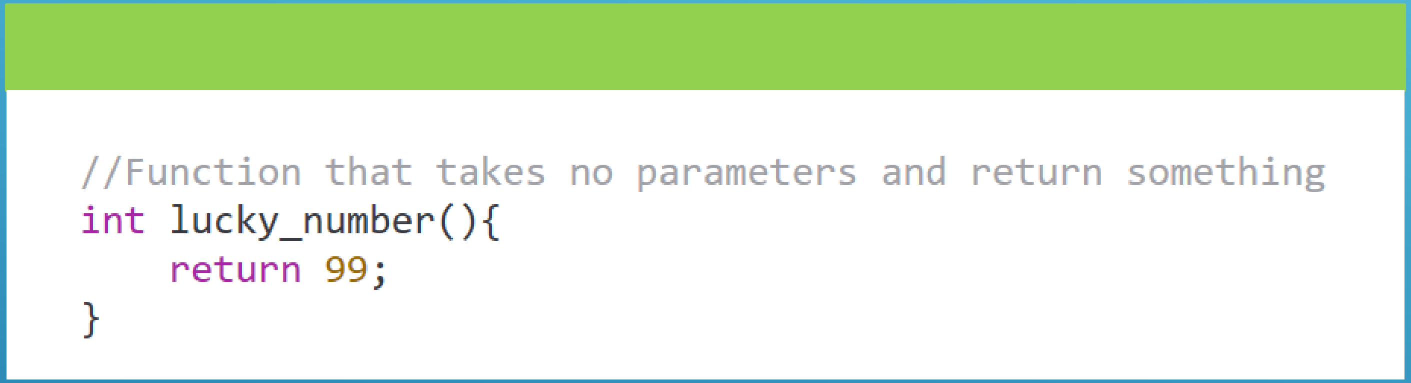
```cpp
//Function that takes a single parameter, and doesn't
//give back the result explicitly

void enter_bar(int age){
    if(age > 18){
        std::cout << "You're " << age << " years old. Please proceed." << std::endl;
    }else{
        std::cout << "Sorry, you're too young for this. No offense! " << std::endl;
    }
}
```



```
//Function that takes multiple parameters and returns the result of the computation
int max( int a, int b){
    if(a>b)
        return a;
    else
        return b;
}
```

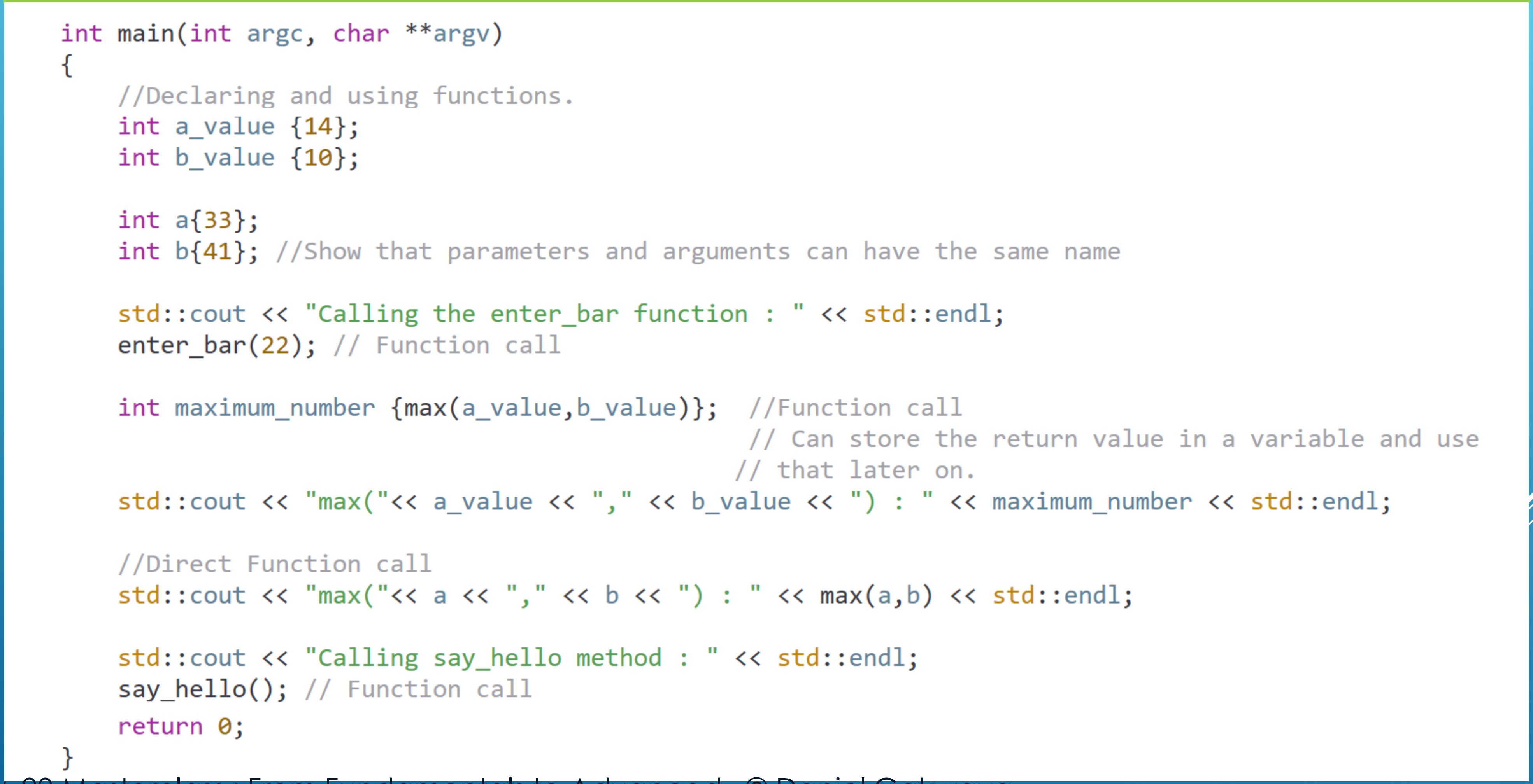


```cpp
//Function that doesnt' take parameters and returns nothing
void say_hello(){
    std::cout << "Hello there" << std::endl;
    return; // You could omit this return statement for functions returning void
}
```



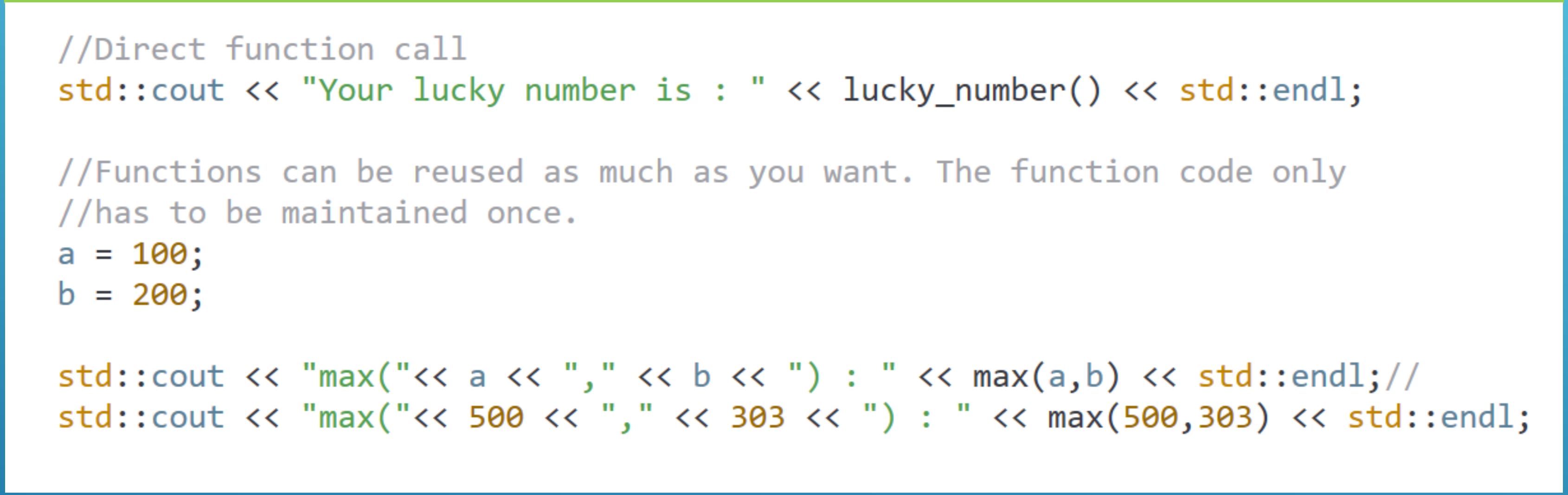
```cpp
//Function that takes no parameters and return something
int lucky_number(){
    return 99;
}
```

Calling Functions



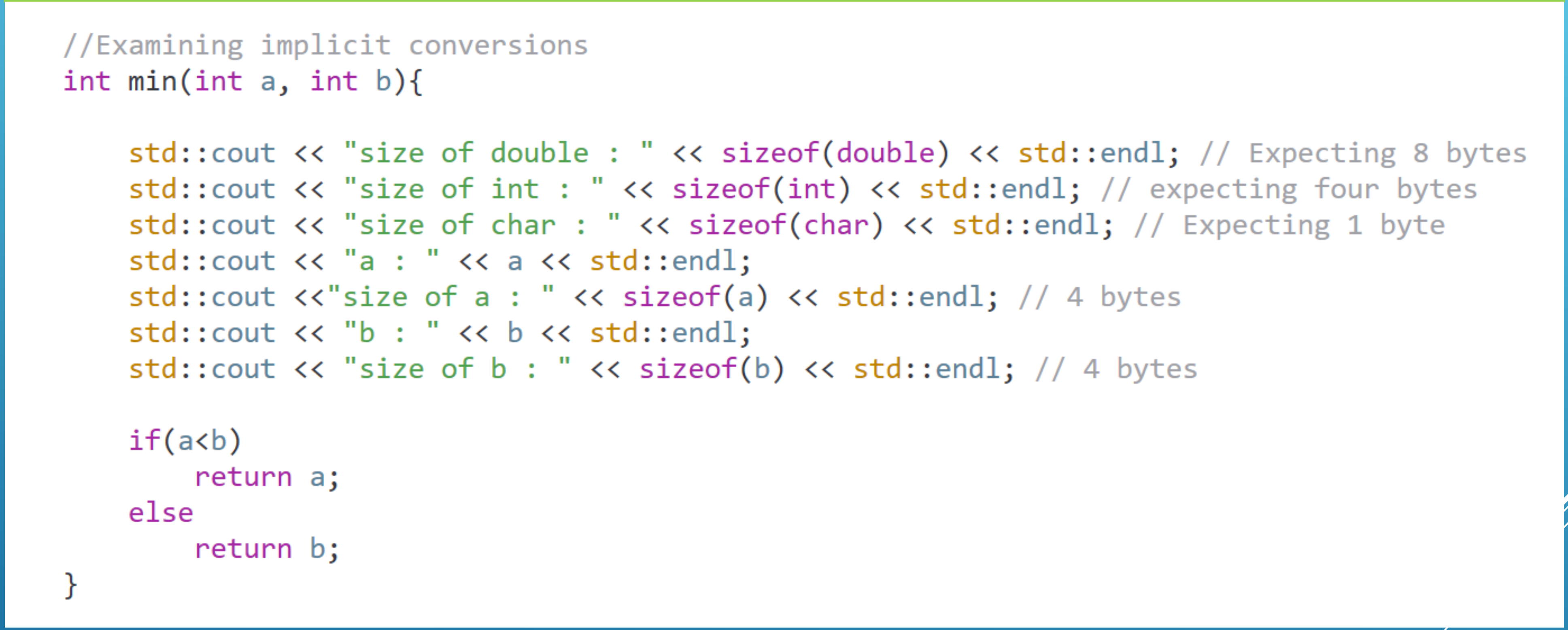
```cpp
int main(int argc, char **argv)
{
    //Declaring and using functions.
    int a_value {14};
    int b_value {10};

    int a{33};
    int b{41}; //Show that parameters and arguments can have the same name

    std::cout << "Calling the enter_bar function : " << std::endl;
    enter_bar(22); // Function call

    int maximum_number {max(a_value,b_value)};  //Function call
                                                // Can store the return value in a variable and use
                                               // that later on.
    std::cout << "max("<< a_value << "," << b_value << ") : " << maximum_number << std::endl;

    //Direct Function call
    std::cout << "max("<< a << "," << b << ") : " << max(a,b) << std::endl;

    std::cout << "Calling say_hello method : " << std::endl;
    say_hello(); // Function call
    return 0;
}
```

Calling Functions



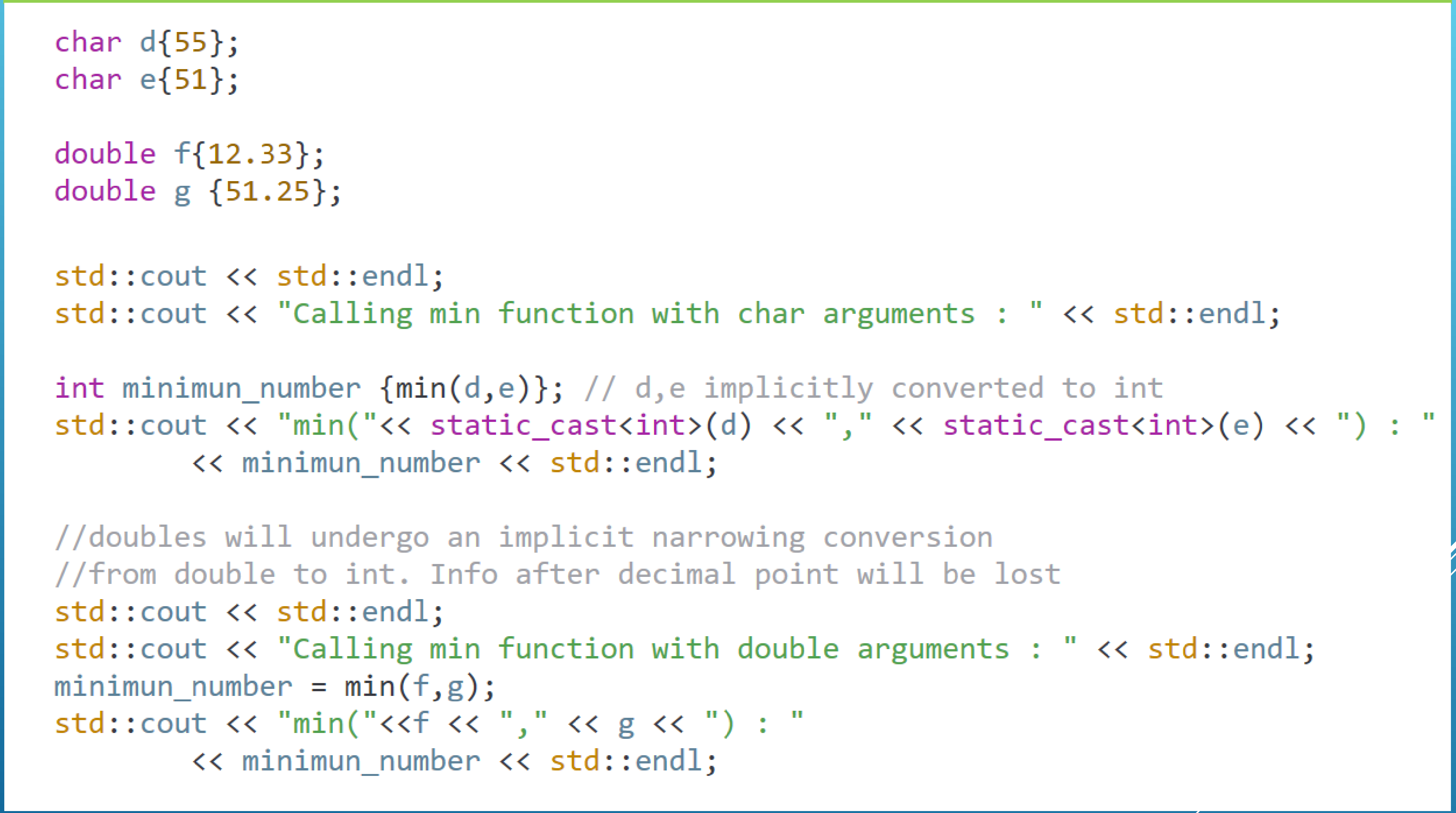
```
//Direct function call
std::cout << "Your lucky number is : " << lucky_number() << std::endl;

//Functions can be reused as much as you want. The function code only
//has to be maintained once.
a = 100;
b = 200;

std::cout << "max("<< a << "," << b << ") : " << max(a,b) << std::endl;//
std::cout << "max("<< 500 << "," << 303 << ") : " << max(500,303) << std::endl;
```

Implicit conversions in functions



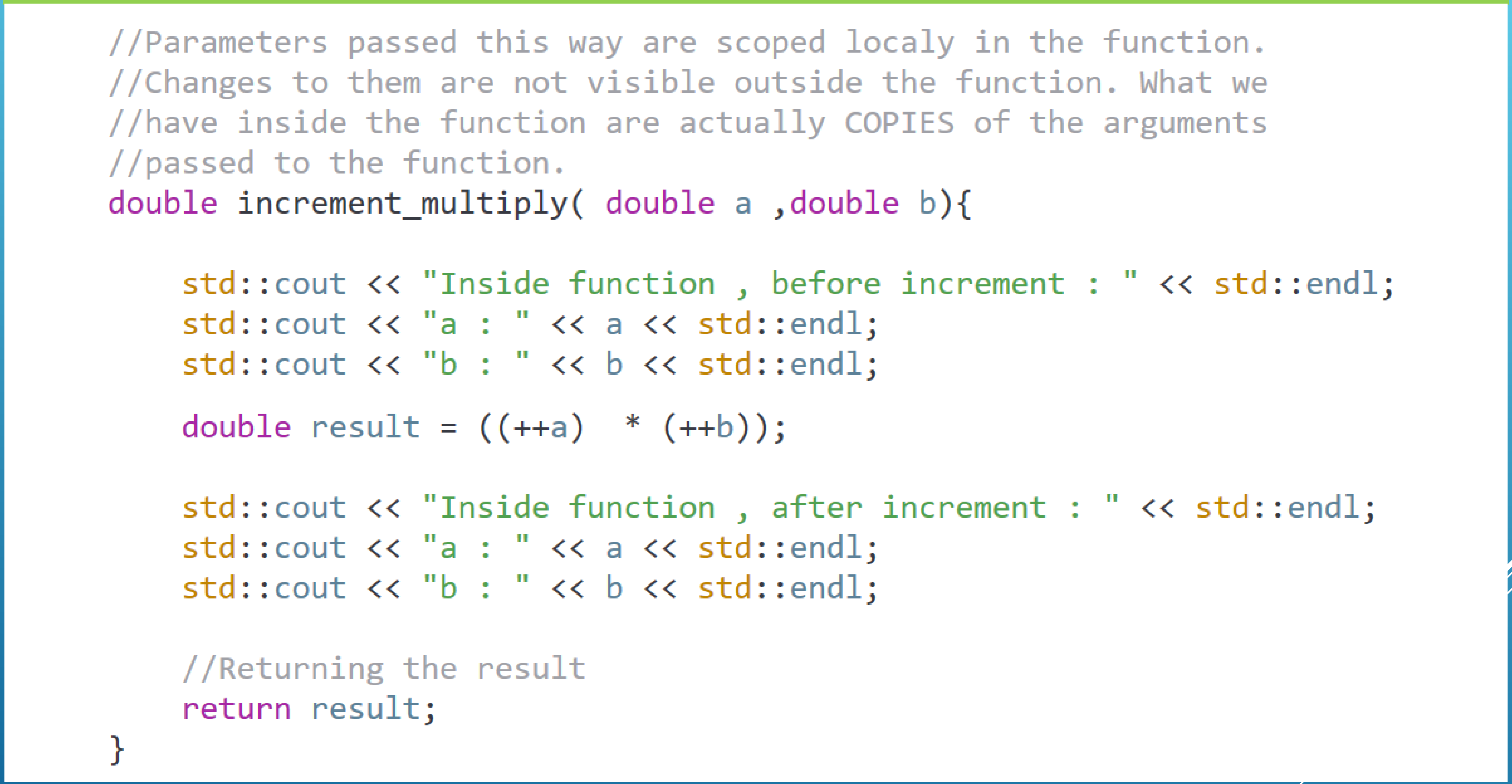
```
//Examining implicit conversions
int min(int a, int b){

    std::cout << "size of double : " << sizeof(double) << std::endl; // Expecting 8 bytes
    std::cout << "size of int : " << sizeof(int) << std::endl; // expecting four bytes
    std::cout << "size of char : " << sizeof(char) << std::endl; // Expecting 1 byte
    std::cout << "a : " << a << std::endl;
    std::cout <<"size of a : " << sizeof(a) << std::endl; // 4 bytes
    std::cout << "b : " << b << std::endl;
    std::cout << "size of b : " << sizeof(b) << std::endl; // 4 bytes

    if(a<b)
        return a;
    else
        return b;
}
```

Implicit conversions in functions



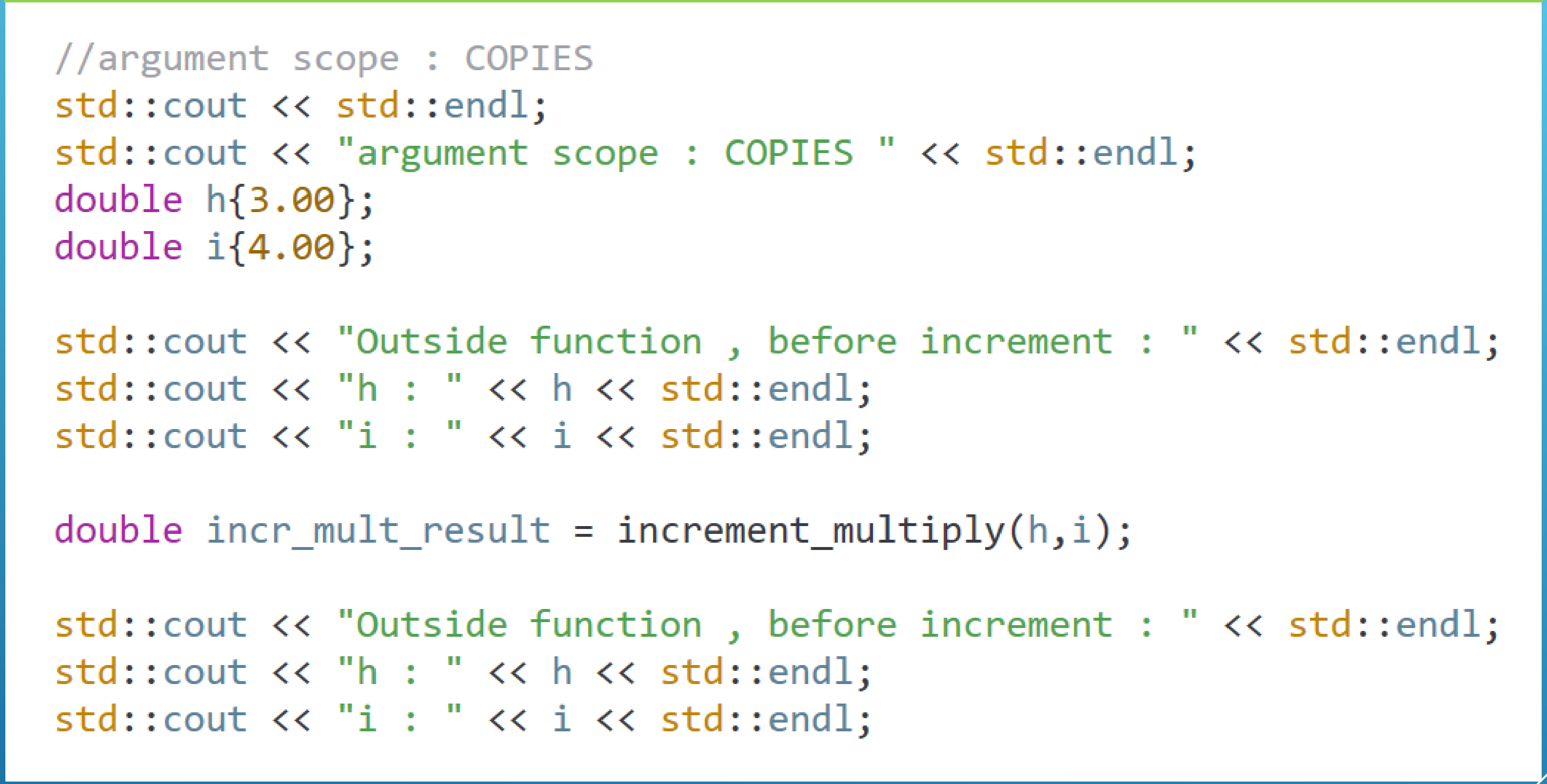
```
char d{55};
char e{51};

double f{12.33};
double g {51.25};

std::cout << std::endl;
std::cout << "Calling min function with char arguments : " << std::endl;

int minimun_number {min(d,e)}; // d,e implicitly converted to int
std::cout << "min("<< static_cast<int>(d) << "," << static_cast<int>(e) << ") : "
        << minimun_number << std::endl;

//doubles will undergo an implicit narrowing conversion
//from double to int. Info after decimal point will be lost
std::cout << std::endl;
std::cout << "Calling min function with double arguments : " << std::endl;
minimun_number = min(f,g);
std::cout << "min("<<f << "," << g << ") : "
        << minimun_number << std::endl;
```

Argument scope : COPIES



```
//Parameters passed this way are scoped localy in the function.
//Changes to them are not visible outside the function. What we
//have inside the function are actually COPIES of the arguments
//passed to the function.
double increment_multiply( double a ,double b){

    std::cout << "Inside function , before increment : " << std::endl;
    std::cout << "a : " << a << std::endl;
    std::cout << "b : " << b << std::endl;

    double result = ((++a)  * (++b));

    std::cout << "Inside function , after increment : " << std::endl;
    std::cout << "a : " << a << std::endl;
    std::cout << "b : " << b << std::endl;

    //Returning the result
    return result;
}
```

Argument scope : COPIES



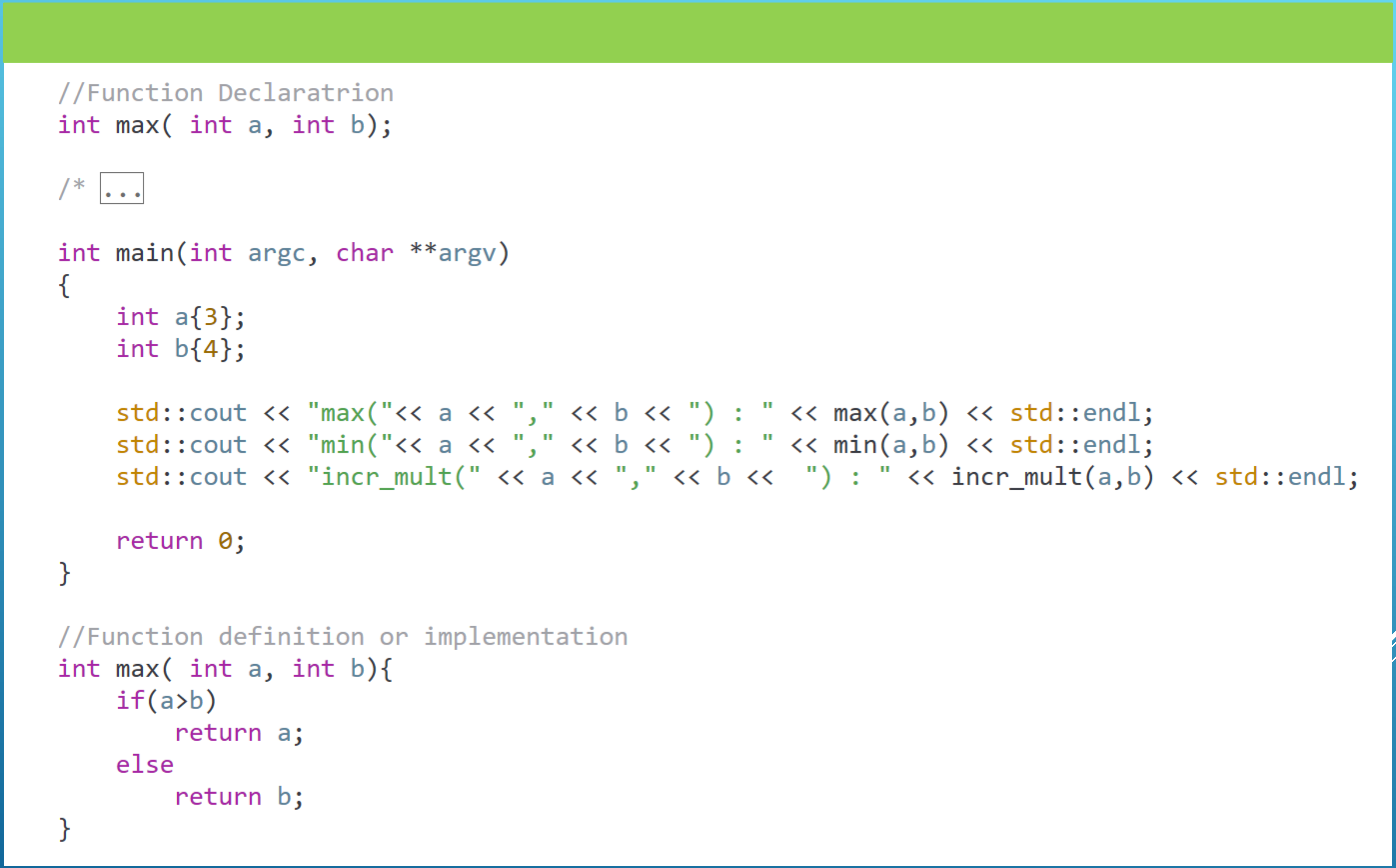
```cpp
//argument scope : COPIES
std::cout << std::endl;
std::cout << "argument scope : COPIES " << std::endl;
double h{3.00};
double i{4.00};

std::cout << "Outside function , before increment : " << std::endl;
std::cout << "h : " << h << std::endl;
std::cout << "i : " << i << std::endl;

double incr_mult_result = increment_multiply(h,i);

std::cout << "Outside function , before increment : " << std::endl;
std::cout << "h : " << h << std::endl;
std::cout << "i : " << i << std::endl;
```

Slide intentionally left empty

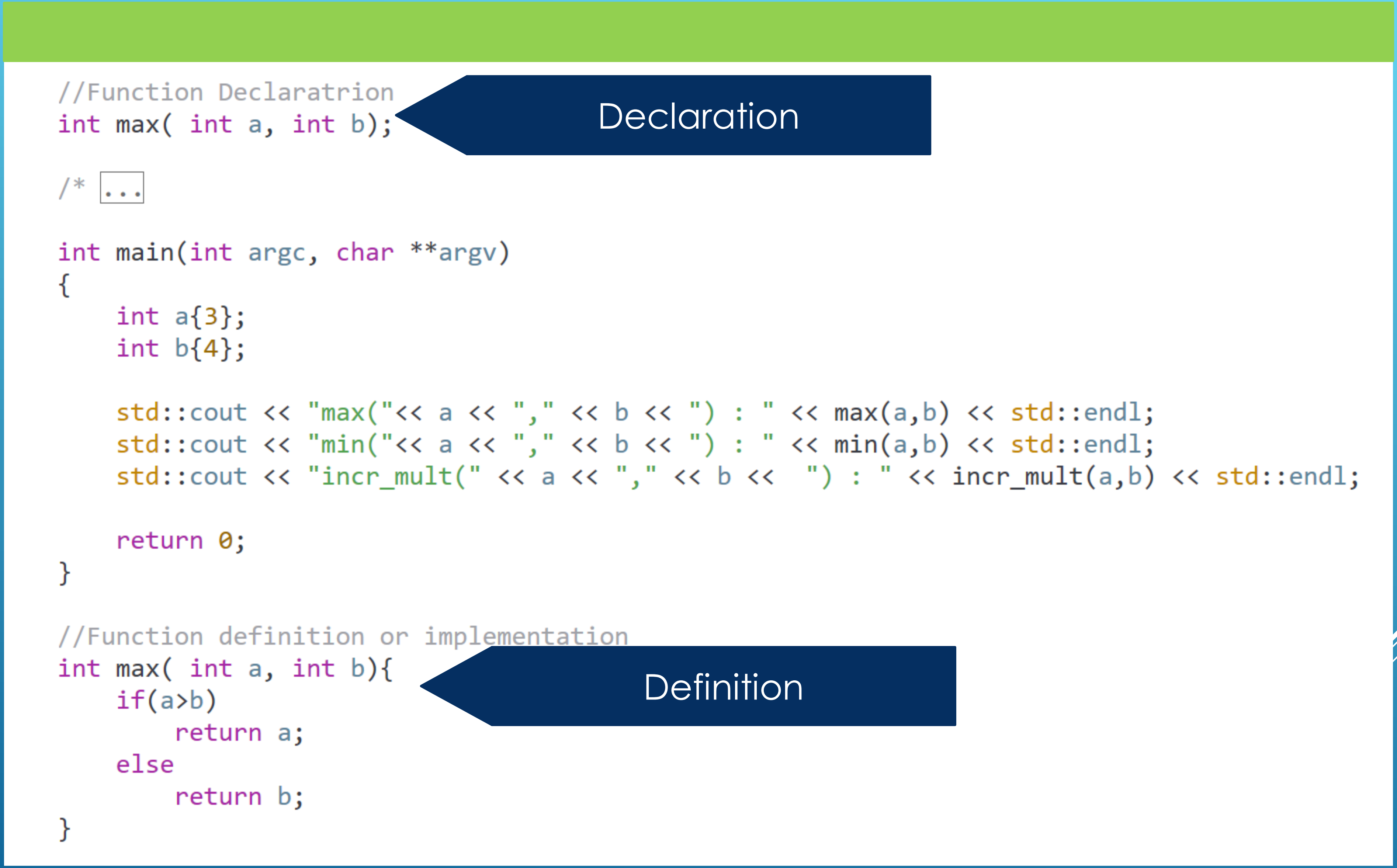
Function Declaration & Definition

Separating stuff

Sometimes it’s more flexible to split the function into it’s header and body and keep the code for each in different places

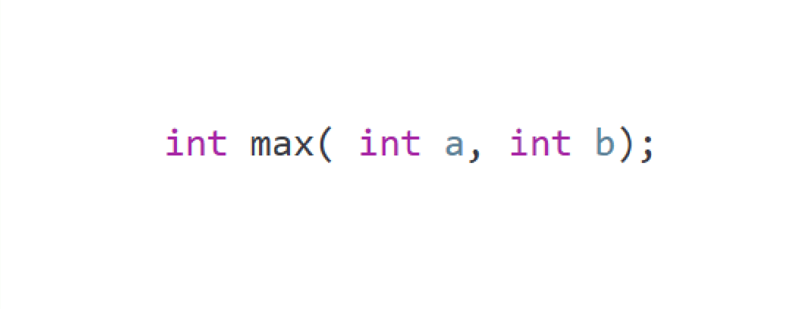


```
//Function Declaratrion
int max( int a, int b);

/* ...

int main(int argc, char **argv)
{
    int a{3};
    int b{4};

    std::cout << "max("<< a << "," << b << ") : " << max(a,b) << std::endl;
    std::cout << "min("<< a << "," << b << ") : " << min(a,b) << std::endl;
    std::cout << "incr_mult(" << a << "," << b <<  ") : " << incr_mult(a,b) << std::endl;

    return 0;
}

//Function definition or implementation
int max( int a, int b){
    if(a>b)
        return a;
    else
        return b;
}
```



```
//Function Declaratrion
int max( int a, int b);

/* ...

int main(int argc, char **argv)
{
    int a{3};
    int b{4};

    std::cout << "max("<< a << "," << b << ") : " << max(a,b) << std::endl;
    std::cout << "min("<< a << "," << b << ") : " << min(a,b) << std::endl;
    std::cout << "incr_mult(" << a << "," << b <<  ") : " << incr_mult(a,b) << std::endl;

    return 0;
}

//Function definition or implementation
int max( int a, int b){
    if(a>b)
        return a;
    else
        return b;
}
```

Prototype



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

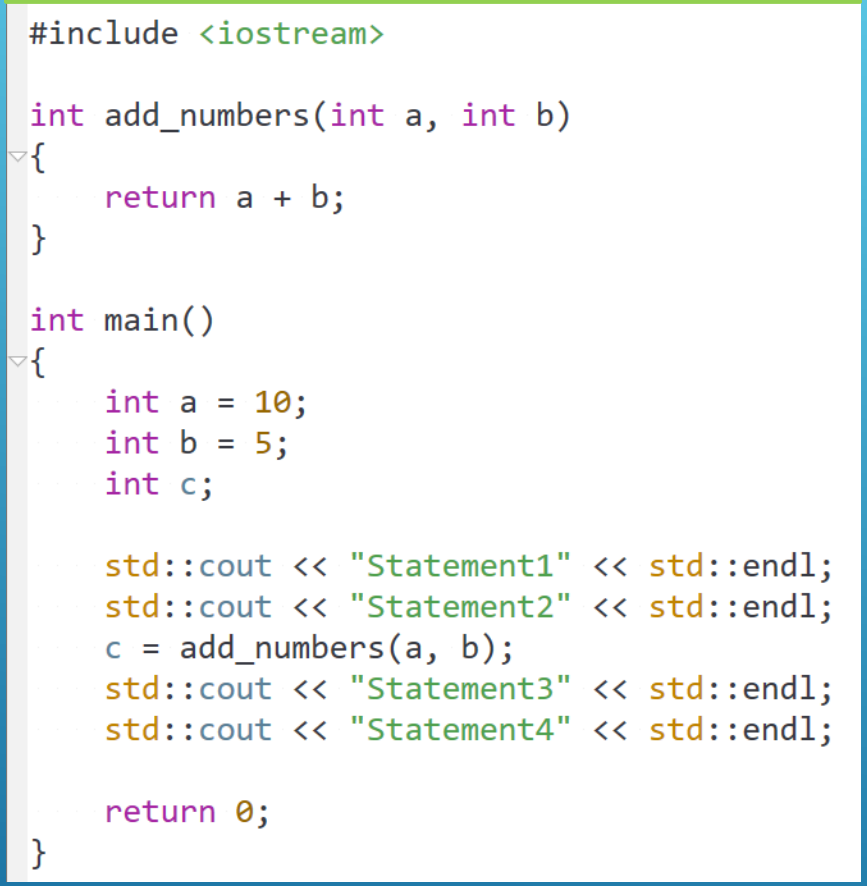
The prototype needs to come BEFORE the function call in your file. Otherwise the compilation will fail.

The full function definition coming in front of main() also doubles as a prototype(declaration). That’s why the code in the last lecture worked without any problem

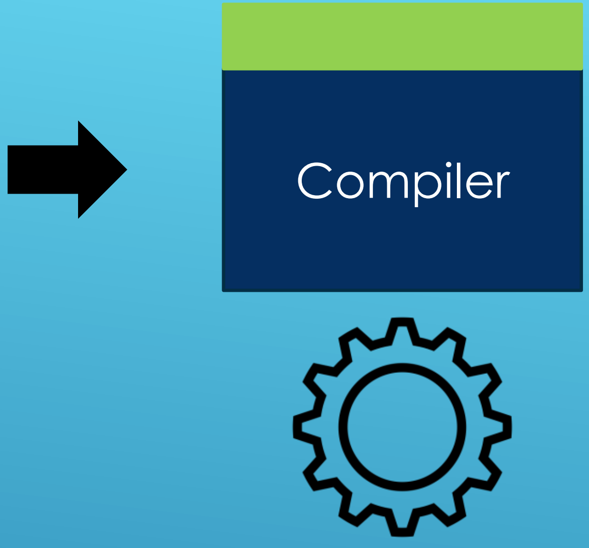
Slide intentionally left empty

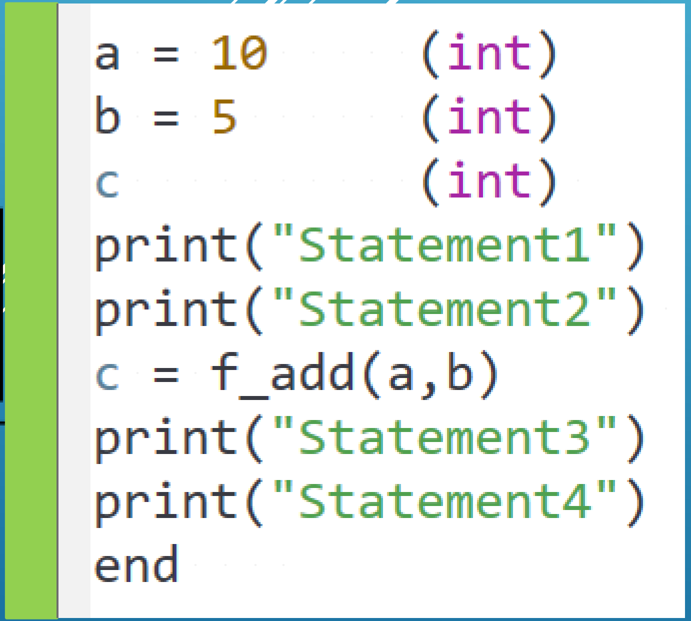
Functions across Multiple Files : Compilation model revisited

One file program



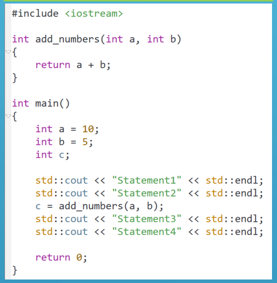
```cpp
#include <iostream>

int add_numbers(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 10;
    int b = 5;
    int c;

    std::cout << "Statement1" << std::endl;
    std::cout << "Statement2" << std::endl;
    c = add_numbers(a, b);
    std::cout << "Statement3" << std::endl;
    std::cout << "Statement4" << std::endl;

    return 0;
}
```





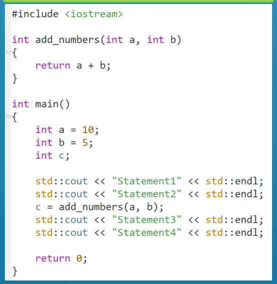
```
a = 10      (int)
b = 5       (int)
c           (int)
print("Statement1")
print("Statement2")
c = f_add(a,b)
print("Statement3")
print("Statement4")
end
```

source



```
#include <iostream>

int add_numbers(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 10;
    int b = 5;
    int c;

    std::cout << "Statement1" << std::endl;
    std::cout << "Statement2" << std::endl;
    c = add_numbers(a, b);
    std::cout << "Statement3" << std::endl;
    std::cout << "Statement4" << std::endl;

    return 0;
}
```

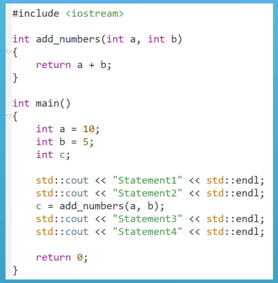
source



```
#include <iostream>

int add_numbers(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 10;
    int b = 5;
    int c;

    std::cout << "Statement1" << std::endl;
    std::cout << "Statement2" << std::endl;
    c = add_numbers(a, b);
    std::cout << "Statement3" << std::endl;
    std::cout << "Statement4" << std::endl;

    return 0;
}
```

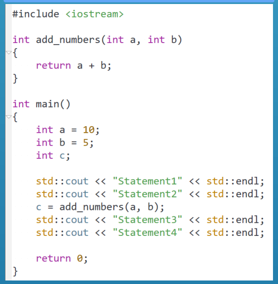
Preprocessing

TU



```
#include <iostream>

int add_numbers(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 10;
    int b = 5;
    int c;

    std::cout << "Statement1" << std::endl;
    std::cout << "Statement2" << std::endl;
    c = add_numbers(a, b);
    std::cout << "Statement3" << std::endl;
    std::cout << "Statement4" << std::endl;

    return 0;
}
```

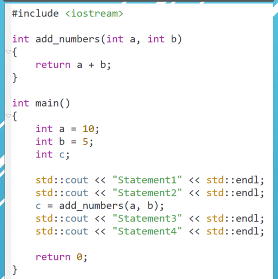
TU



```
#include <iostream>

int add_numbers(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 10;
    int b = 5;
    int c;

    std::cout << "Statement1" << std::endl;
    std::cout << "Statement2" << std::endl;
    c = add_numbers(a, b);
    std::cout << "Statement3" << std::endl;
    std::cout << "Statement4" << std::endl;

    return 0;
}
```

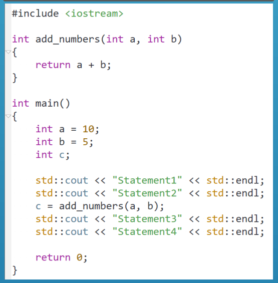
Compilation

object



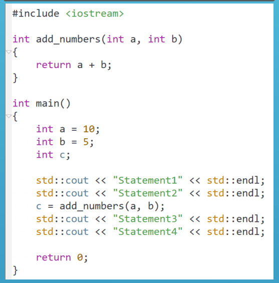
```
#include <iostream>

int add_numbers(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 10;
    int b = 5;
    int c;

    std::cout << "Statement1" << std::endl;
    std::cout << "Statement2" << std::endl;
    c = add_numbers(a, b);
    std::cout << "Statement3" << std::endl;
    std::cout << "Statement4" << std::endl;

    return 0;
}
```

object



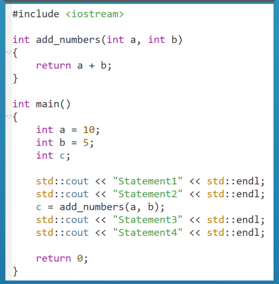
```
#include <iostream>

int add_numbers(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 10;
    int b = 5;
    int c;

    std::cout << "Statement1" << std::endl;
    std::cout << "Statement2" << std::endl;
    c = add_numbers(a, b);
    std::cout << "Statement3" << std::endl;
    std::cout << "Statement4" << std::endl;

    return 0;
}
```

object



```
#include <iostream>

int add_numbers(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 10;
    int b = 5;
    int c;

    std::cout << "Statement1" << std::endl;
    std::cout << "Statement2" << std::endl;
    c = add_numbers(a, b);
    std::cout << "Statement3" << std::endl;
    std::cout << "Statement4" << std::endl;

    return 0;
}
```

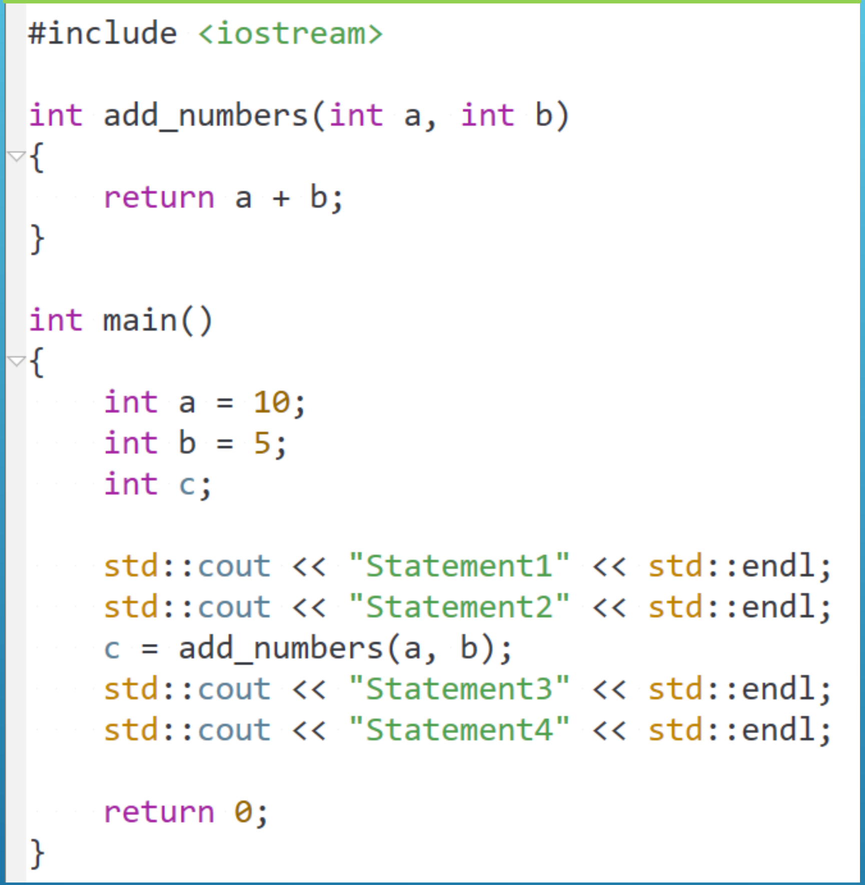
object



```
#include <iostream>

int add_numbers(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 10;
    int b = 5;
    int c;

    std::cout << "Statement1" << std::endl;
    std::cout << "Statement2" << std::endl;
    c = add_numbers(a, b);
    std::cout << "Statement3" << std::endl;
    std::cout << "Statement4" << std::endl;

    return 0;
}
```

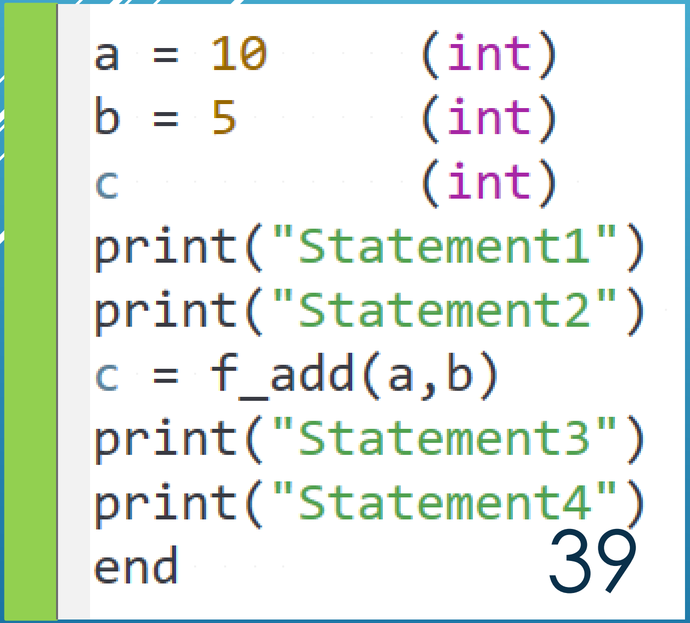
Linking

One file program



```cpp
#include <iostream>

int add_numbers(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 10;
    int b = 5;
    int c;

    std::cout << "Statement1" << std::endl;
    std::cout << "Statement2" << std::endl;
    c = add_numbers(a, b);
    std::cout << "Statement3" << std::endl;
    std::cout << "Statement4" << std::endl;

    return 0;
}
```

- Preprocessing
- Compilation
- Linking



```
a = 10      (int)
b = 5       (int)
c           (int)
print("Statement1")
print("Statement2")
c = f_add(a,b)
print("Statement3")
print("Statement4")
end
```

comparisons.h

int max(int a, int b);
int min(int a, int b);

operations.h

int incr_mult(int a, int b);

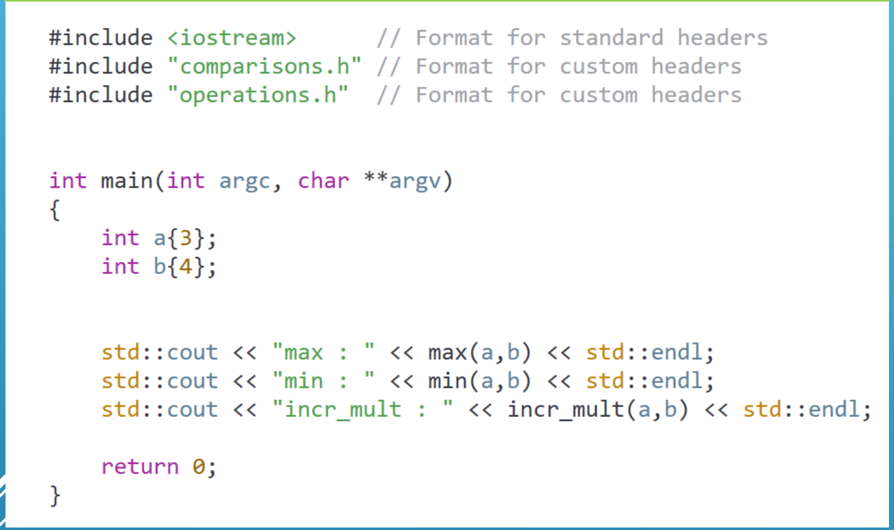
comparisons.cpp

implemenation

operations.cpp

implementation

Main file



```
#include <iostream>       // Format for standard headers
#include "comparisons.h" // Format for custom headers
#include "operations.h"  // Format for custom headers


int main(int argc, char **argv)
{
    int a{3};
    int b{4};


    std::cout << "max : " << max(a,b) << std::endl;
    std::cout << "min : " << min(a,b) << std::endl;
    std::cout << "incr_mult : " << incr_mult(a,b) << std::endl;

    return 0;
}
```

ODR

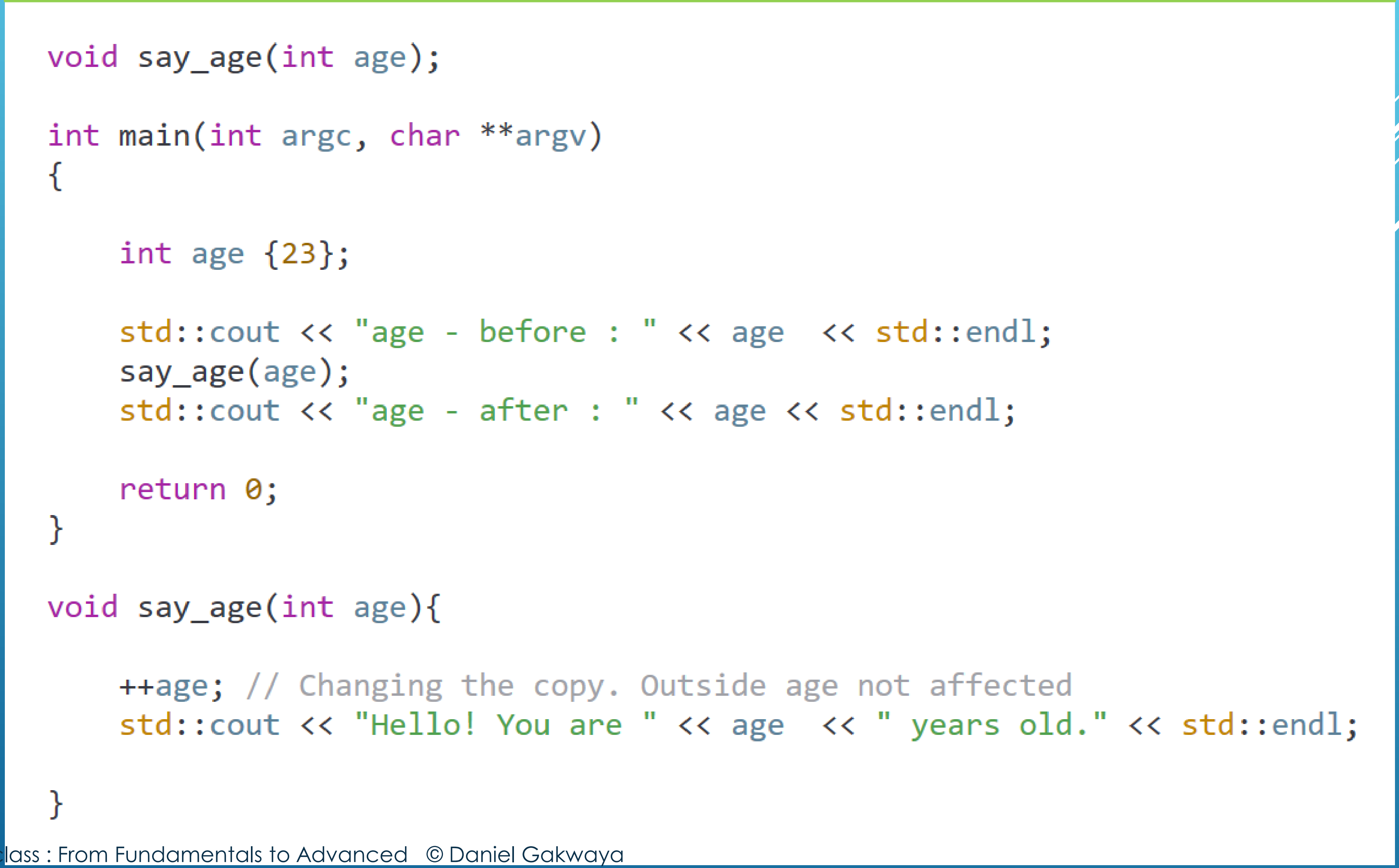
One Definition Rule : The same function implementation can’t show up in the global namespace more than once.

The linker searches for definitions in all translation units (.cpp) files in the project. Doesn’t have to live in a .cpp file with the same name as the header

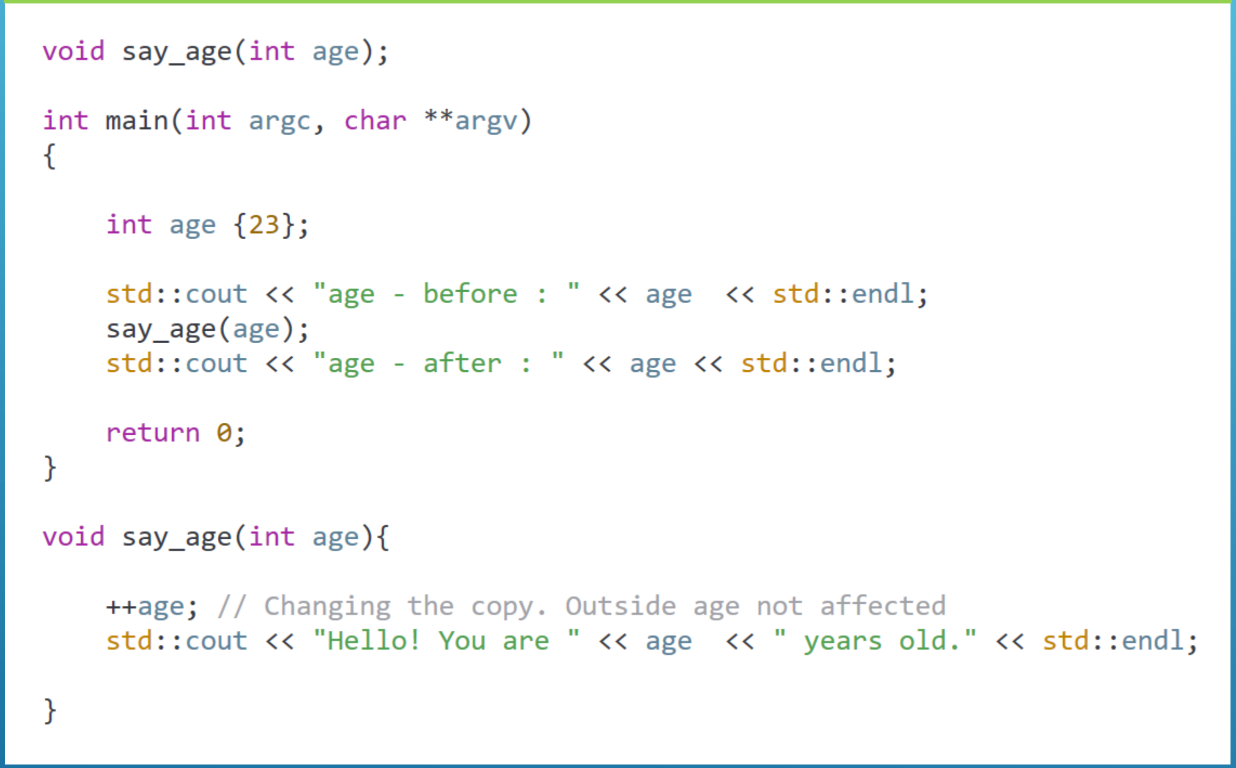
Slide intentionally left empty

Pass by value

Pass by value : syntax



```
void say_age(int age);

int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    say_age(age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(int age){

    ++age; // Changing the copy. Outside age not affected
    std::cout << "Hello! You are " << age  << " years old." << std::endl;

}
```



```
void say_age(int age);

int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    say_age(age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(int age){

    ++age; // Changing the copy. Outside age not affected
    std::cout << "Hello! You are " << age  << " years old." << std::endl;

}
```

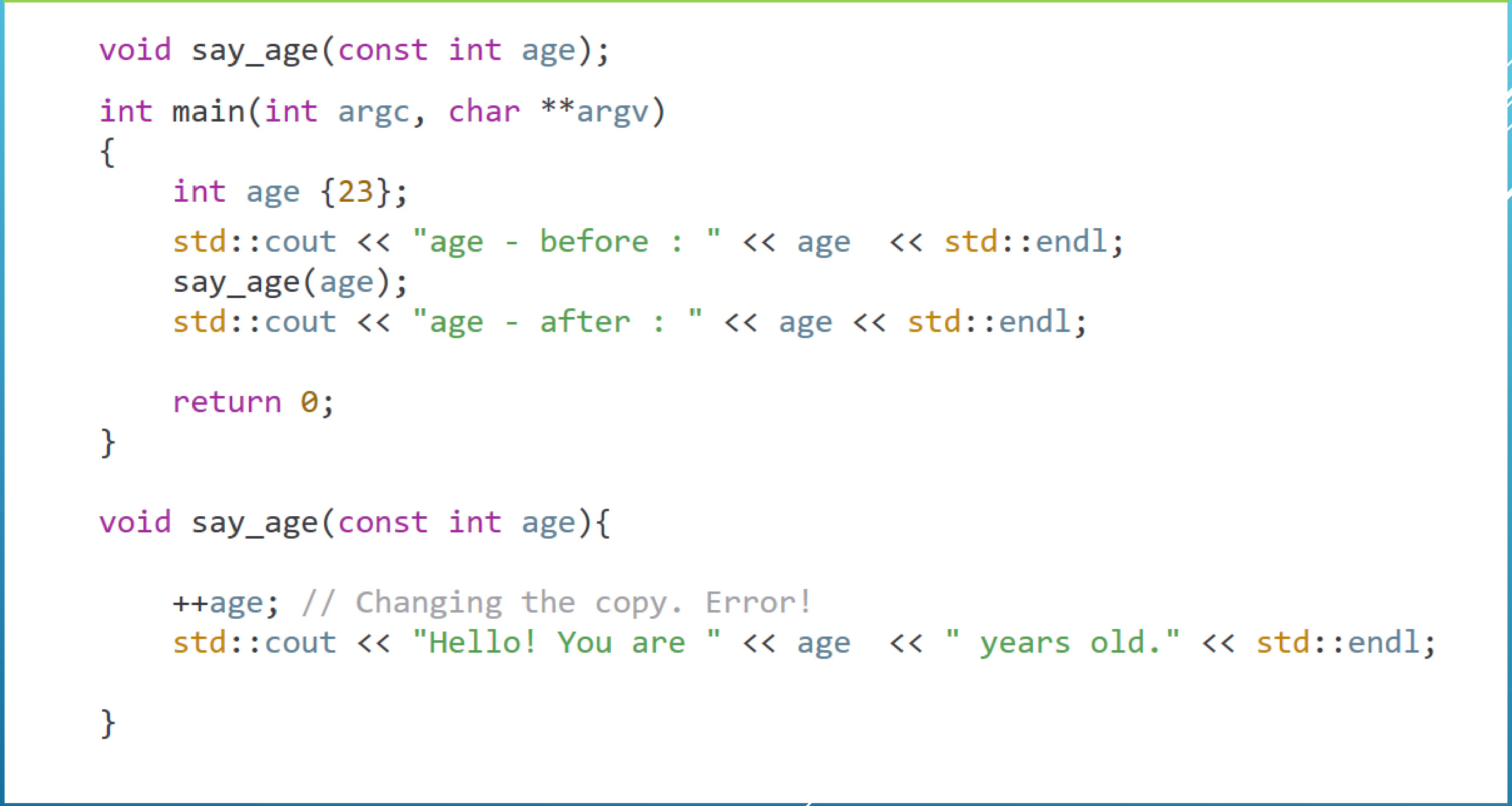
say_age() function call context

```
{

    int age_copy {age_value};
    //use age_copy

    ...
    //age copy goes out of scope

}
```

Slide intentionally left empty

Pass by const value

Pass by const value : syntax

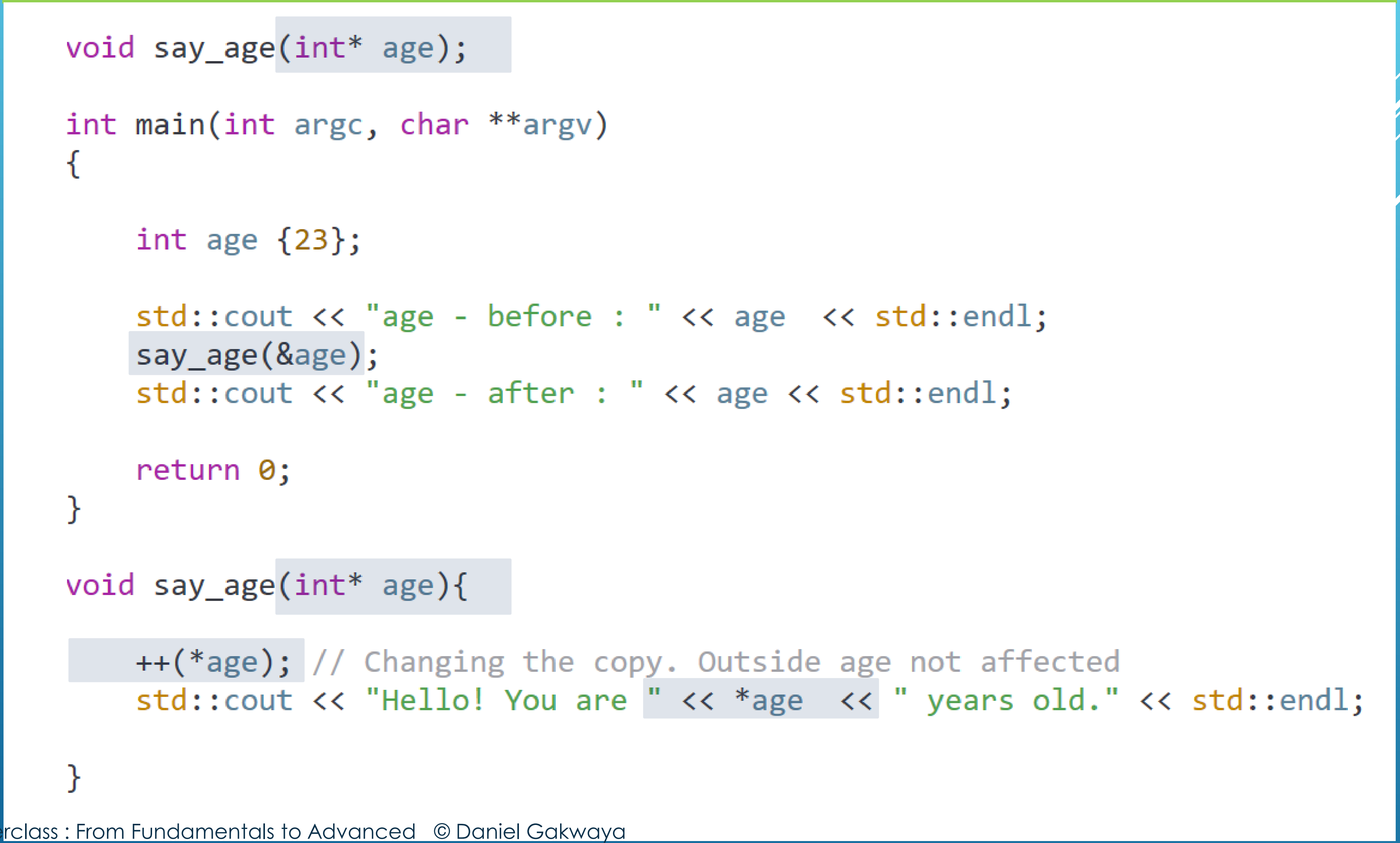


```cpp
void say_age(const int age);

int main(int argc, char **argv)
{
    int age {23};
    std::cout << "age - before : " << age  << std::endl;
    say_age(age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(const int age){

    ++age; // Changing the copy. Error!
    std::cout << "Hello! You are " << age  << " years old." << std::endl;

}
```

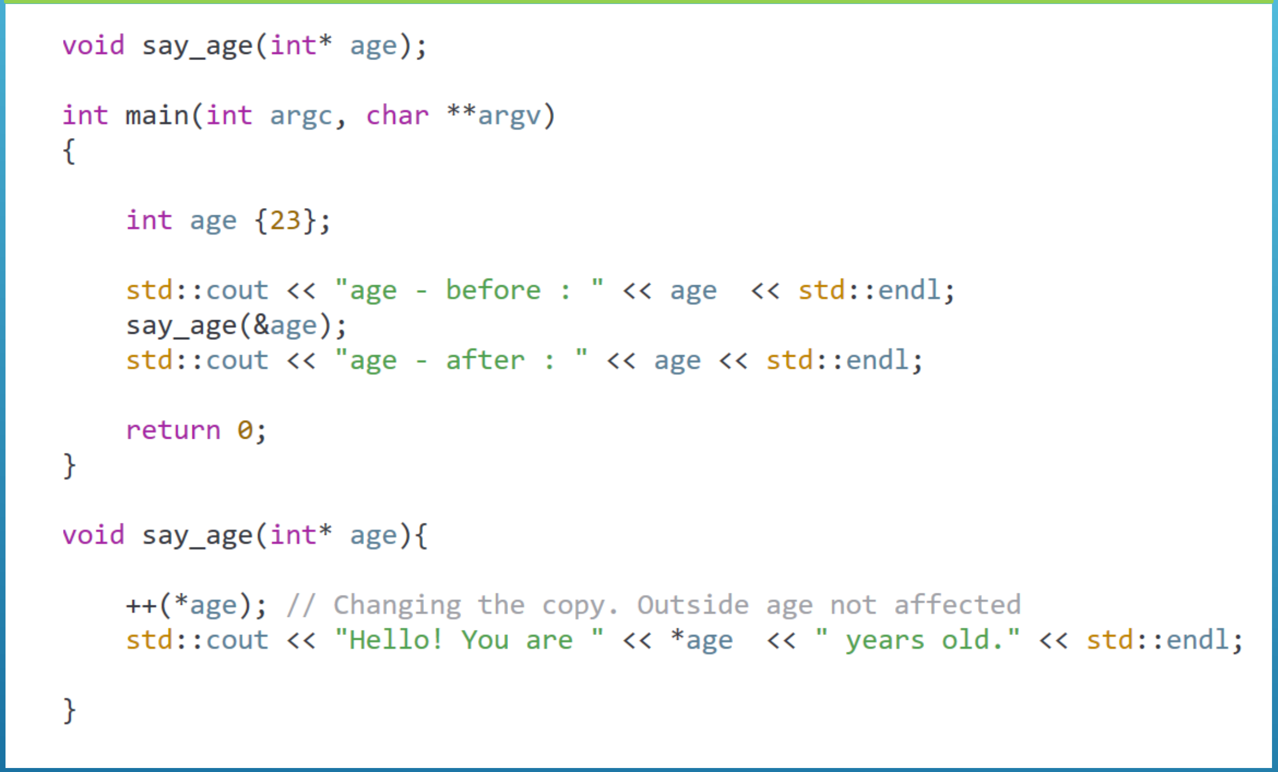
Slide intentionally left empty

Pass by pointer

Pass by pointer : syntax



```cpp
void say_age(int* age);

int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    say_age(&age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(int* age){

    ++(*age); // Changing the copy. Outside age not affected
    std::cout << "Hello! You are " << *age  << " years old." << std::endl;

}
```



```cpp
void say_age(int* age);

int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    say_age(&age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(int* age){

    ++(*age); // Changing the copy. Outside age not affected
    std::cout << "Hello! You are " << *age  << " years old." << std::endl;

}
```

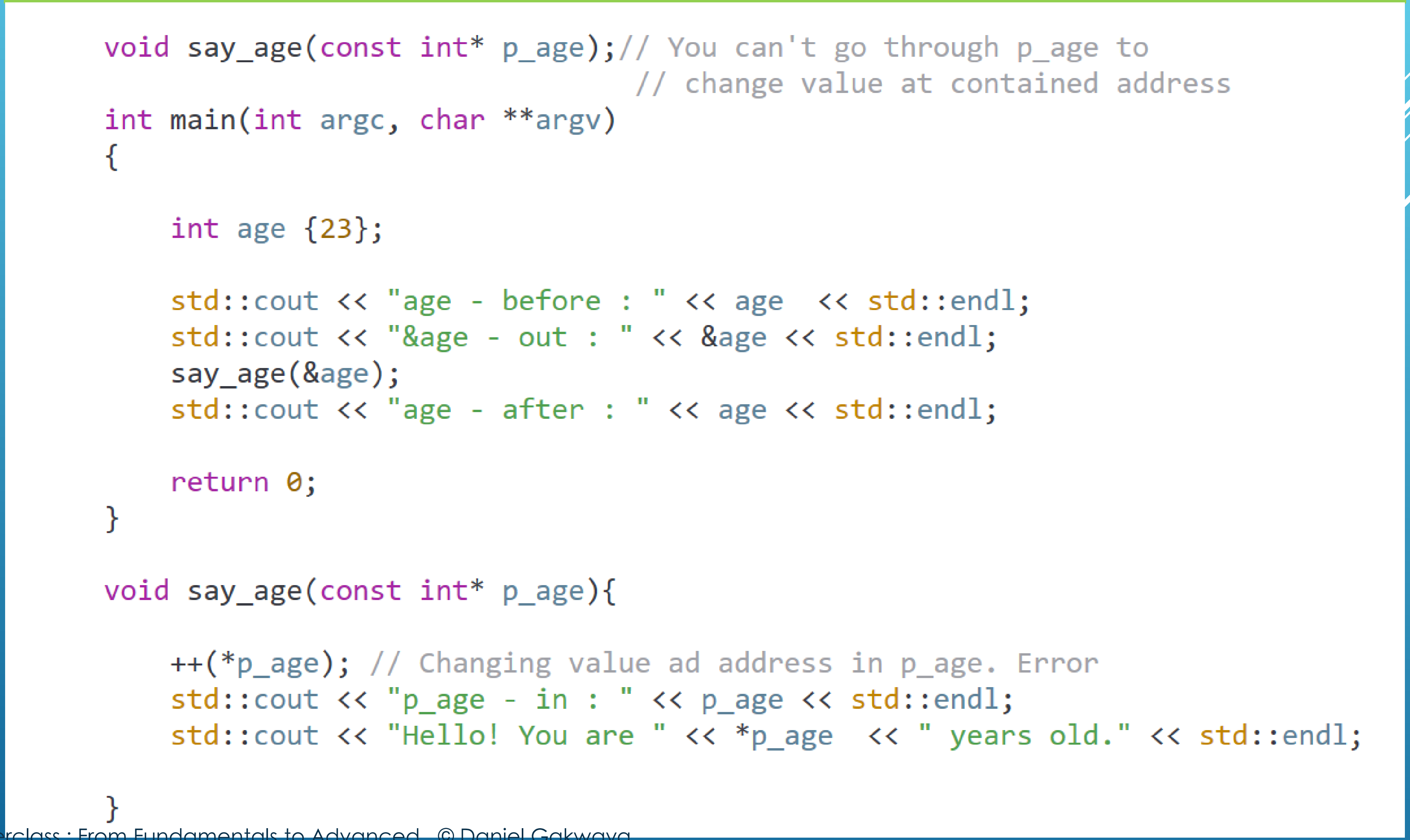
say_age() function call context

```
{

    const int *age_copy {age_address};
    //use age_address

    ...
    //age_address goes out of scope

}
```

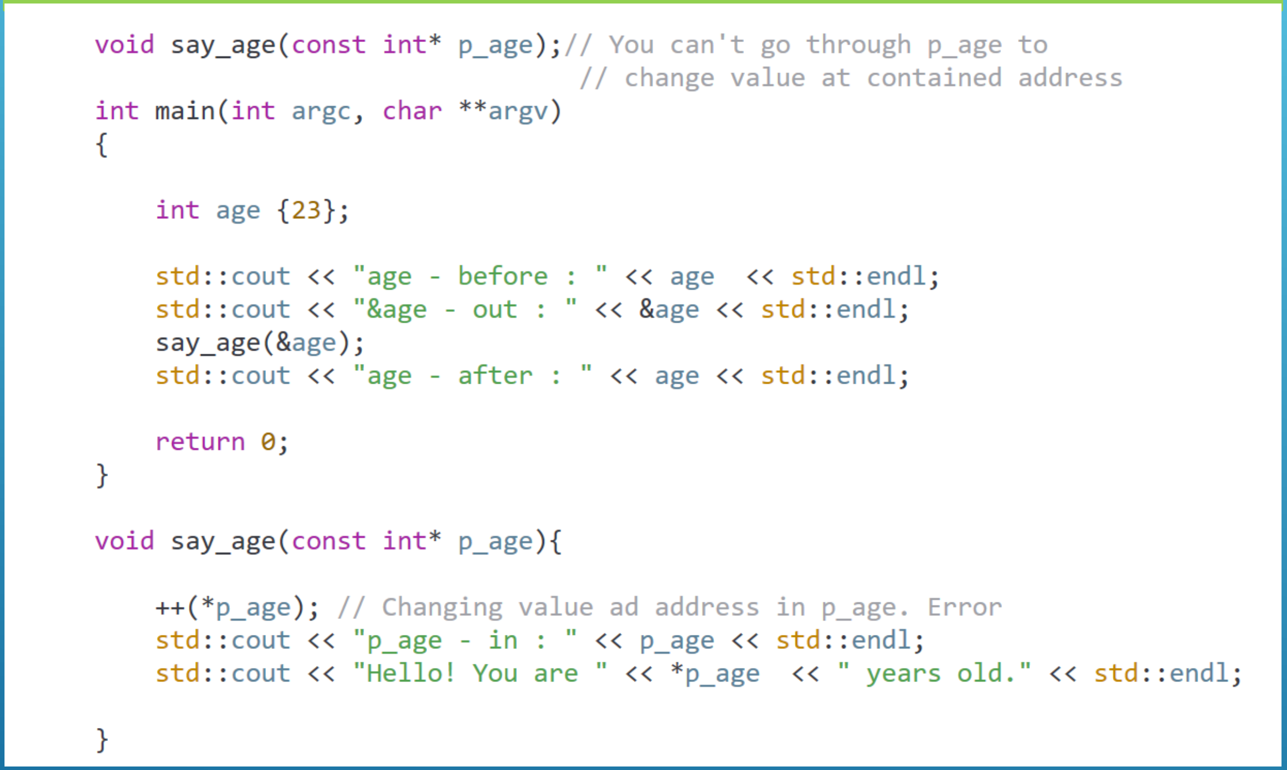
Slide intentionally left empty

Pass by pointer to const

Pass by pointer : syntax



```cpp
void say_age(const int* p_age);// You can't go through p_age to
                                // change value at contained address
int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    std::cout << "&age - out : " << &age << std::endl;
    say_age(&age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(const int* p_age){

    ++(*p_age); // Changing value ad address in p_age. Error
    std::cout << "p_age - in : " << p_age << std::endl;
    std::cout << "Hello! You are " << *p_age  << " years old." << std::endl;

}
```



```cpp
void say_age(const int* p_age);// You can't go through p_age to
                               // change value at contained address
int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    std::cout << "&age - out : " << &age << std::endl;
    say_age(&age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(const int* p_age){

    ++(*p_age); // Changing value ad address in p_age. Error
    std::cout << "p_age - in : " << p_age << std::endl;
    std::cout << "Hello! You are " << *p_age  << " years old." << std::endl;

}
```

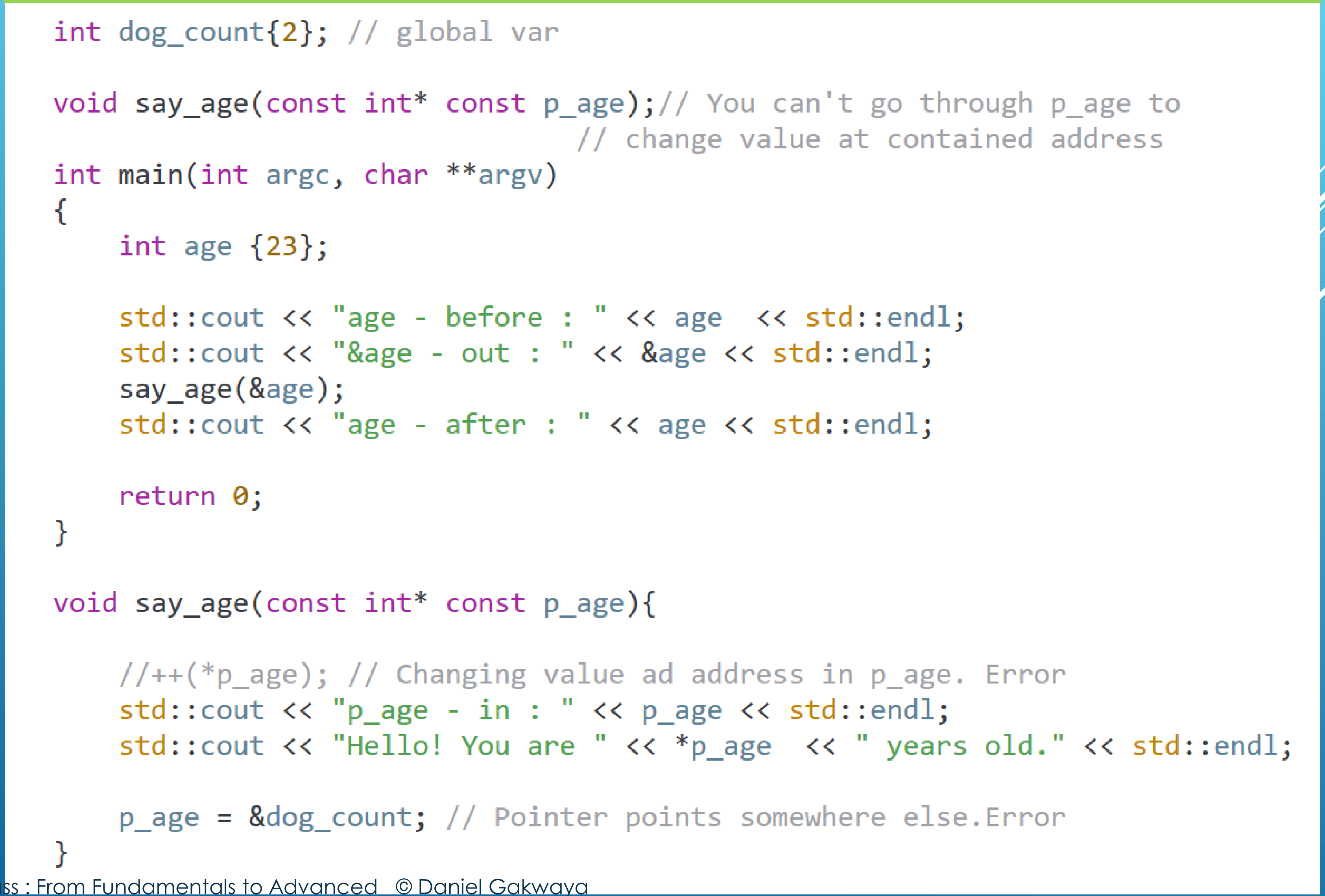
say_age() function call context

```
{

    const int *age_copy {age_address};
    //use age_address

    …
    //age_address goes out of scope

}
```

Slide intentionally left empty

Pass by const pointer to const

Pass by const pointer to const : syntax

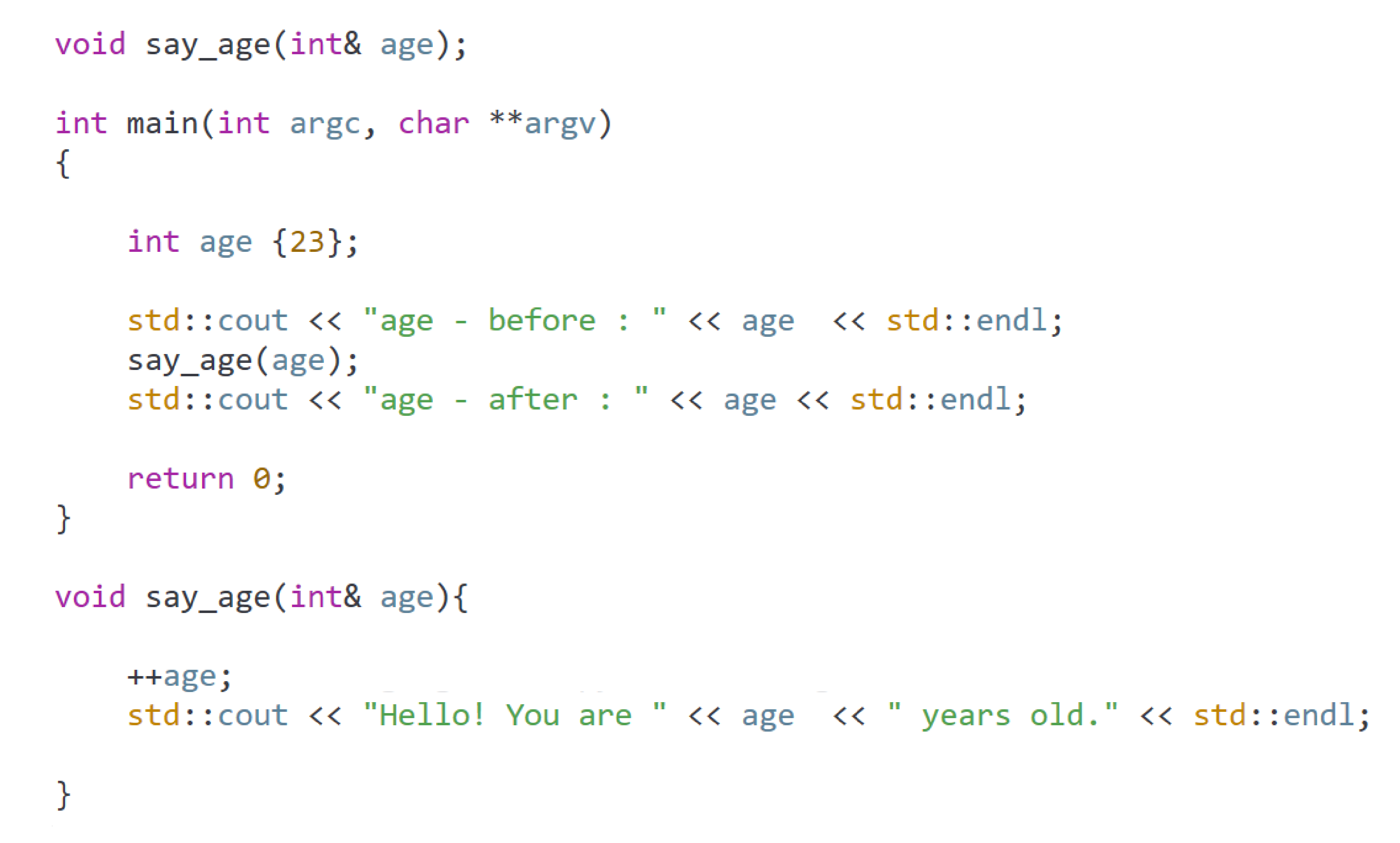


```cpp
int dog_count{2}; // global var

void say_age(const int* const p_age);// You can't go through p_age to
                                // change value at contained address
int main(int argc, char **argv)
{
    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    std::cout << "&age - out : " << &age << std::endl;
    say_age(&age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(const int* const p_age){

    //++(*p_age); // Changing value ad address in p_age. Error
    std::cout << "p_age - in : " << p_age << std::endl;
    std::cout << "Hello! You are " << *p_age  << " years old." << std::endl;

    p_age = &dog_count; // Pointer points somewhere else.Error
}
```

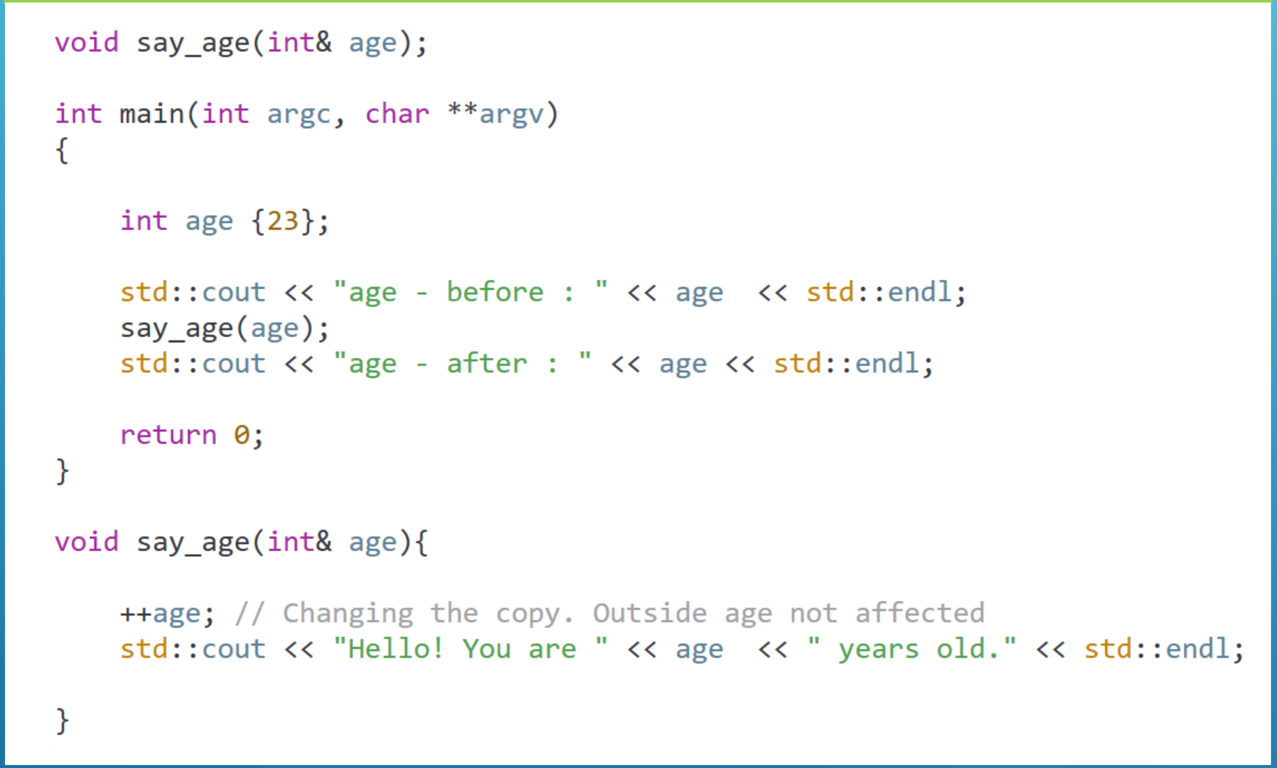
Slide intentionally left empty

Pass by reference

Pass by reference : syntax



```
void say_age(int& age);

int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    say_age(age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(int& age){

    ++age;
    std::cout << "Hello! You are " << age  << " years old." << std::endl;

}
```



```cpp
void say_age(int& age);

int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    say_age(age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(int& age){

    ++age; // Changing the copy. Outside age not affected
    std::cout << "Hello! You are " << age  << " years old." << std::endl;

}
```

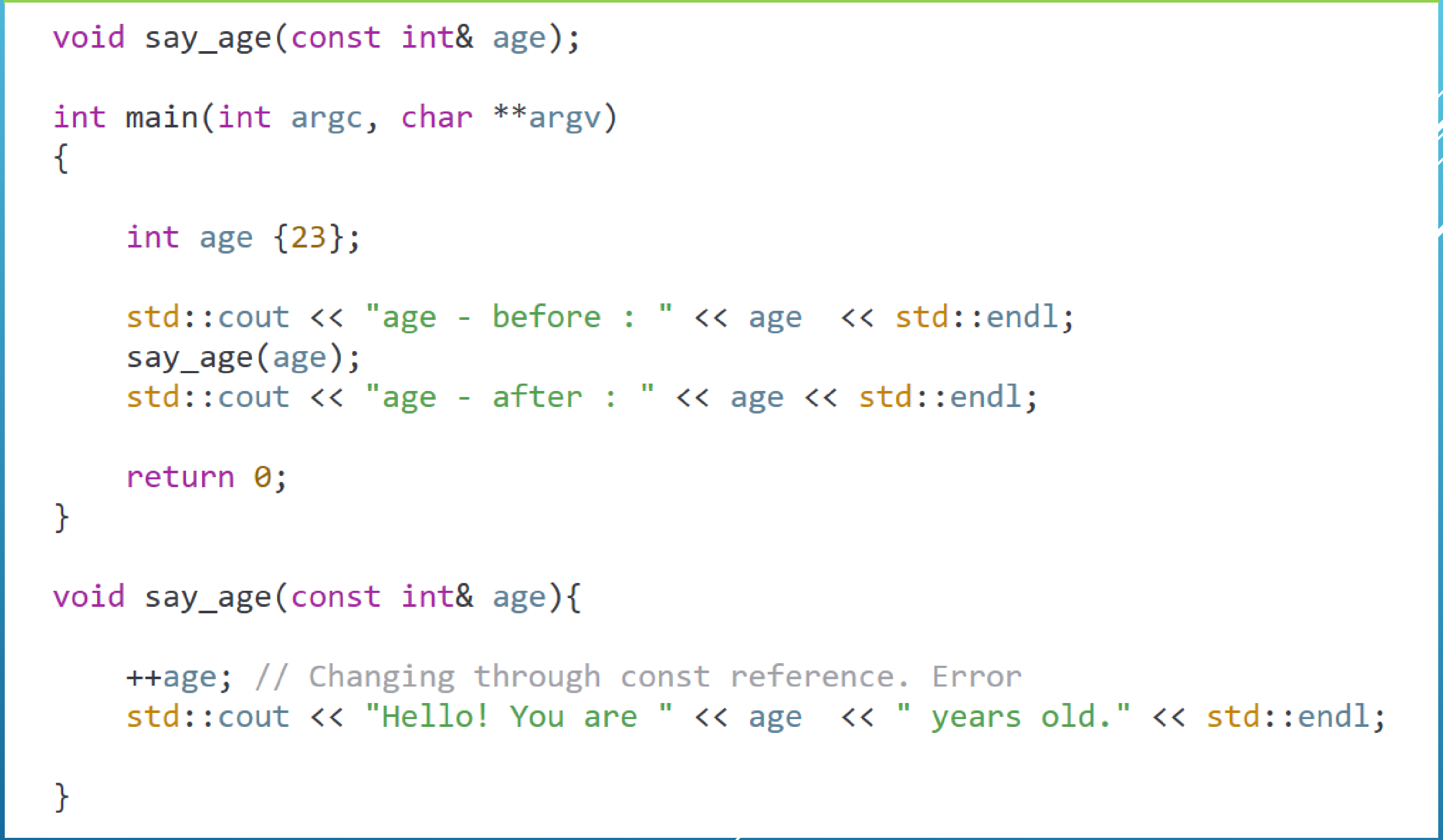
say_age() function call context

```
{

        int &age_ref{age_arg};
        //use age_ref

        ...
        //age_ref goes out of scope

}
```

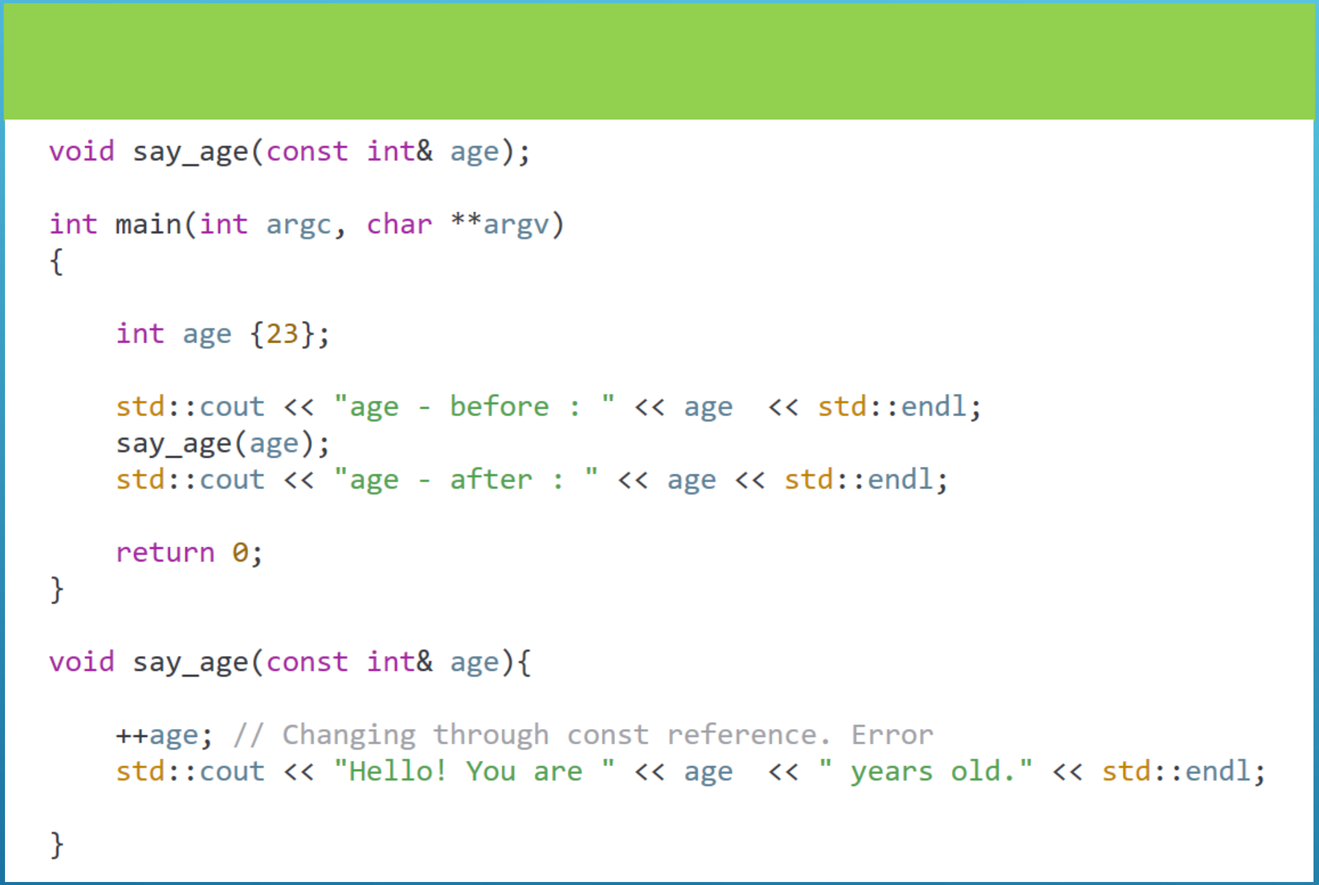
Slide intentionally left empty

Pass by const reference

Pass by const reference : syntax



```cpp
void say_age(const int& age);

int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    say_age(age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(const int& age){

    ++age; // Changing through const reference. Error
    std::cout << "Hello! You are " << age  << " years old." << std::endl;

}
```



```cpp
void say_age(const int& age);

int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    say_age(age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(const int& age){

    ++age; // Changing through const reference. Error
    std::cout << "Hello! You are " << age  << " years old." << std::endl;

}
```

say_age() function call context

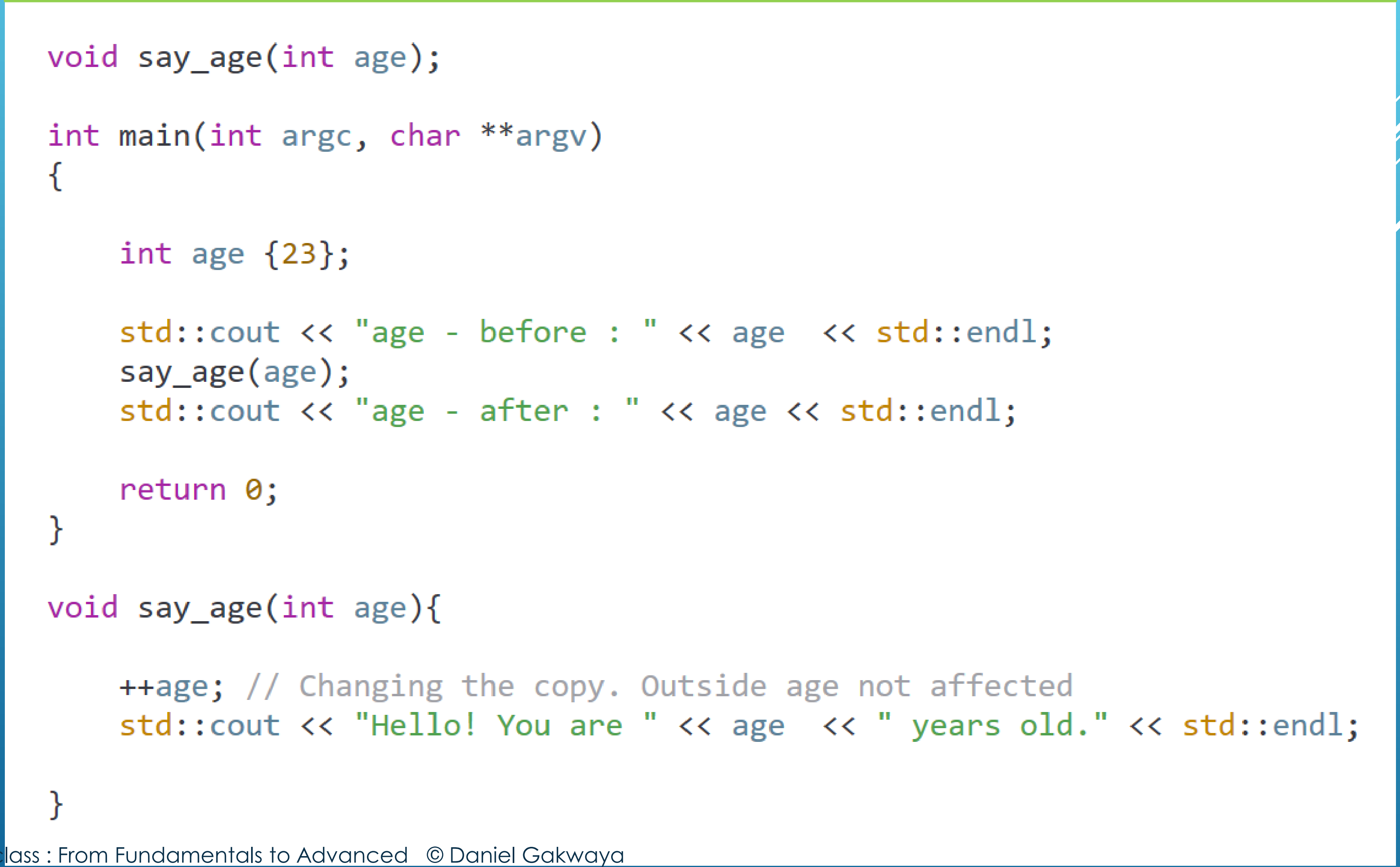
```
{

        int age_copy {age_value};
        //use age_copy

        ...
        //age copy goes out of scope

}
```

Slide intentionally left empty

Passing function parameters : A summary

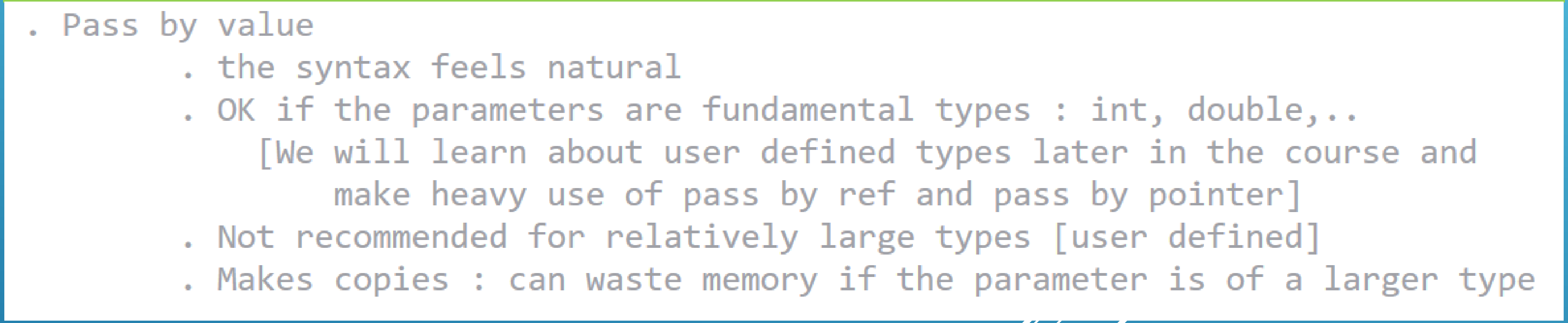
- Pass by value
- Pass by pointer
- Pass by reference

Pass by value



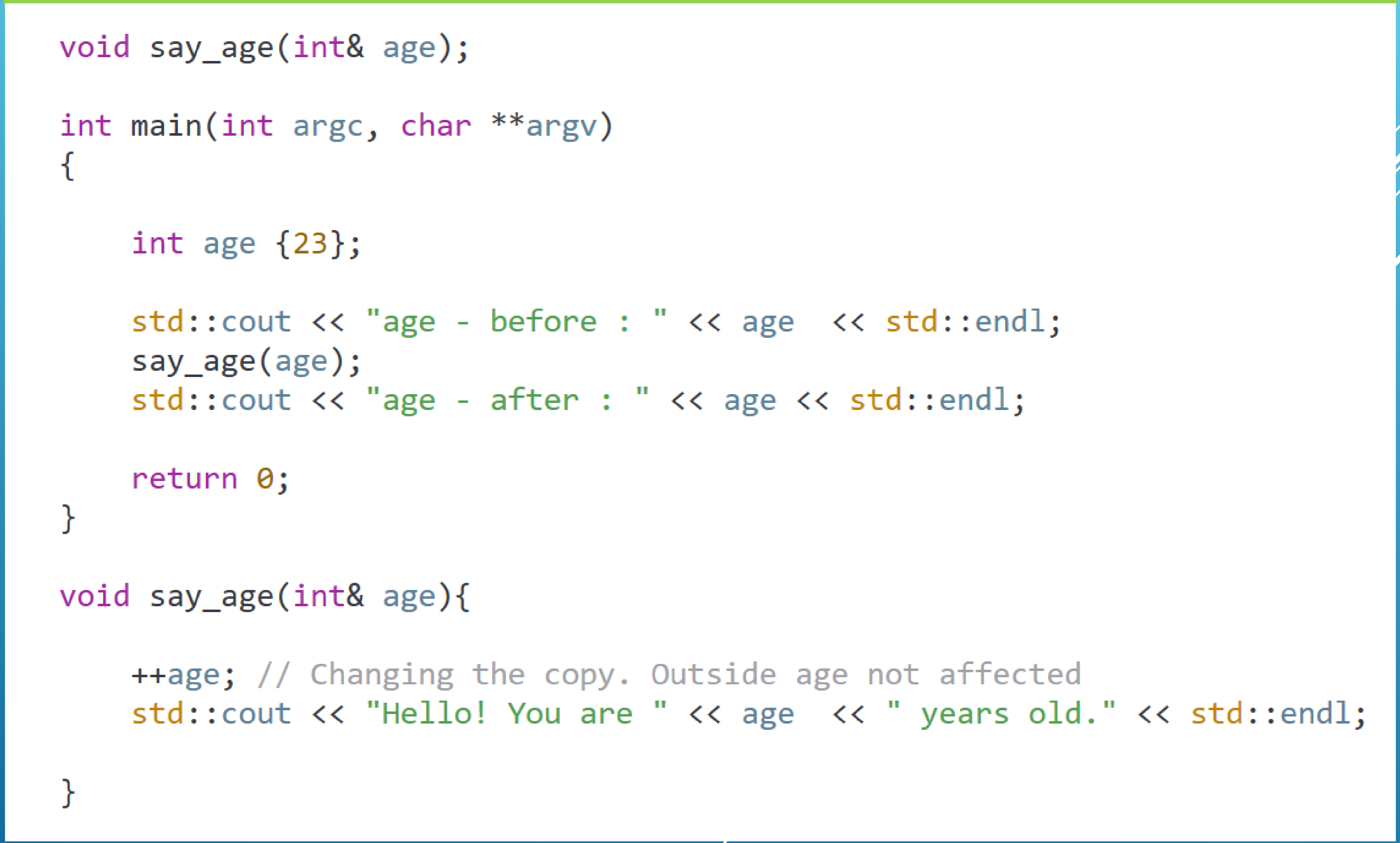
```cpp
void say_age(int age);

int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    say_age(age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(int age){

    ++age; // Changing the copy. Outside age not affected
    std::cout << "Hello! You are " << age  << " years old." << std::endl;

}
```

Pass by value



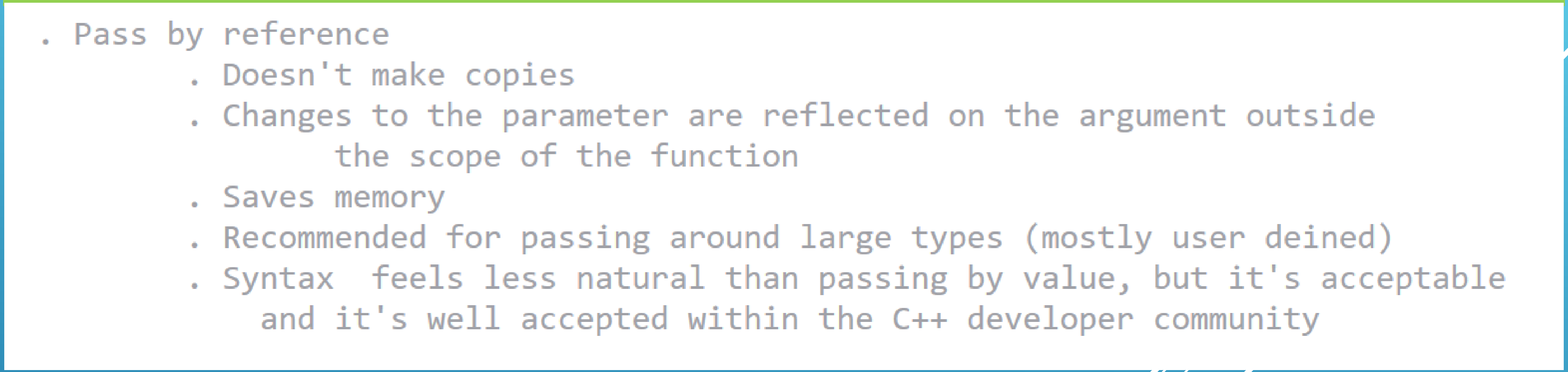
```
. Pass by value
        . the syntax feels natural
        . OK if the parameters are fundamental types : int, double,..
          [We will learn about user defined types later in the course and
              make heavy use of pass by ref and pass by pointer]
        . Not recommended for relatively large types [user defined]
        . Makes copies : can waste memory if the parameter is of a larger type
```

Pass by reference



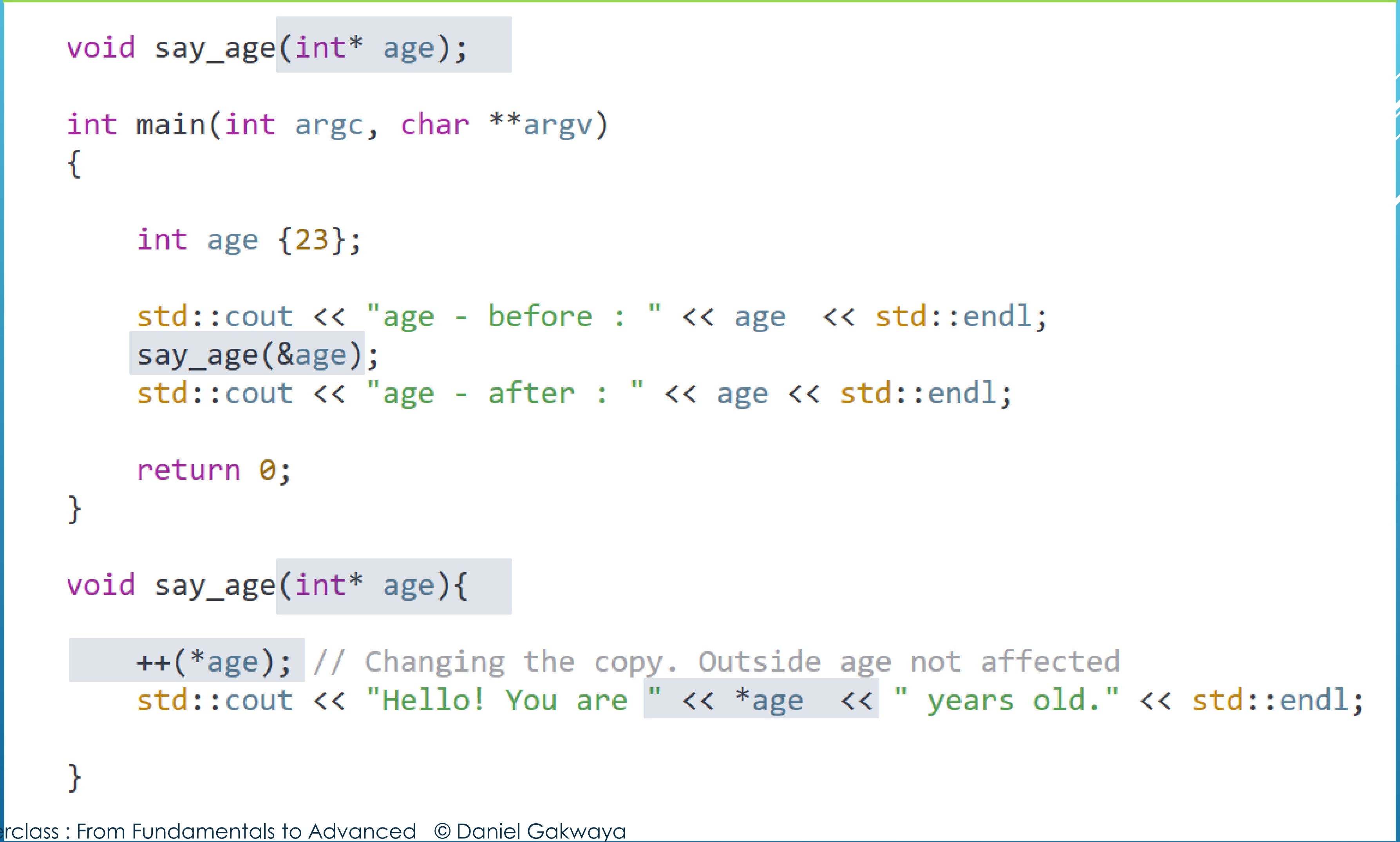
```cpp
void say_age(int& age);

int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    say_age(age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(int& age){

    ++age; // Changing the copy. Outside age not affected
    std::cout << "Hello! You are " << age  << " years old." << std::endl;

}
```

Pass by reference



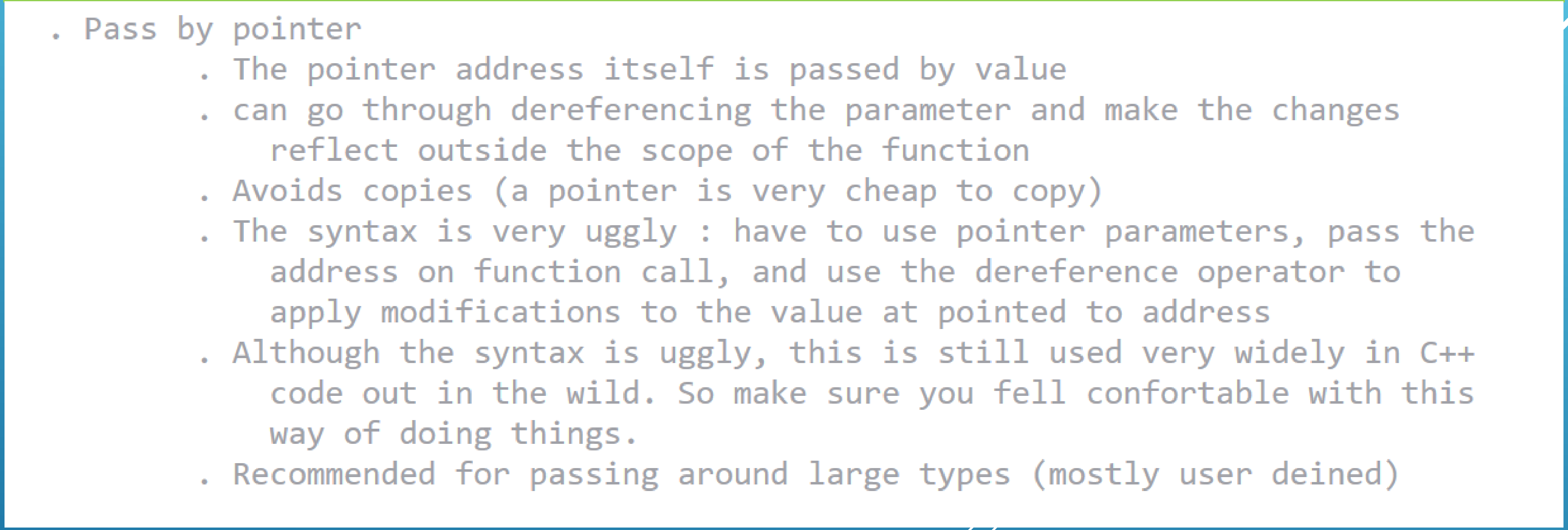
```
. Pass by reference
        . Doesn't make copies
        . Changes to the parameter are reflected on the argument outside
              the scope of the function
        . Saves memory
        . Recommended for passing around large types (mostly user deined)
        . Syntax  feels less natural than passing by value, but it's acceptable
          and it's well accepted within the C++ developer community
```

Pass by pointer



```
void say_age(int* age);

int main(int argc, char **argv)
{

    int age {23};

    std::cout << "age - before : " << age  << std::endl;
    say_age(&age);
    std::cout << "age - after : " << age << std::endl;

    return 0;
}

void say_age(int* age){

    ++(*age); // Changing the copy. Outside age not affected
    std::cout << "Hello! You are " << *age  << " years old." << std::endl;

}
```

Pass by pointer

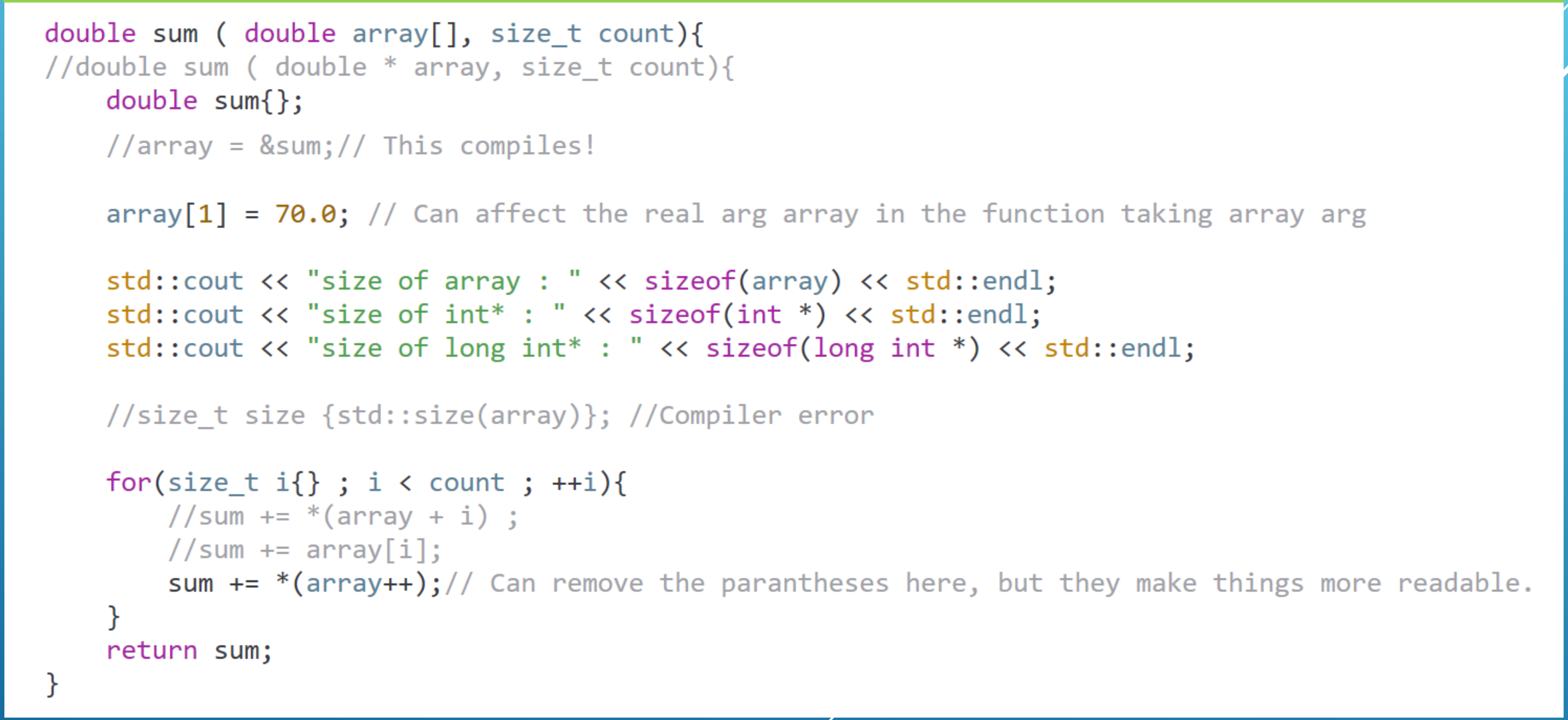


- Pass by pointer
  - The pointer address itself is passed by value
  - can go through dereferencing the parameter and make the changes reflect outside the scope of the function
  - Avoids copies (a pointer is very cheap to copy)
  - The syntax is very uggly : have to use pointer parameters, pass the address on function call, and use the dereference operator to apply modifications to the value at pointed to address
  - Although the syntax is uggly, this is still used very widely in C++ code out in the wild. So make sure you fell confortable with this way of doing things.
  - Recommended for passing around large types (mostly user deined)

Slide intentionally left empty

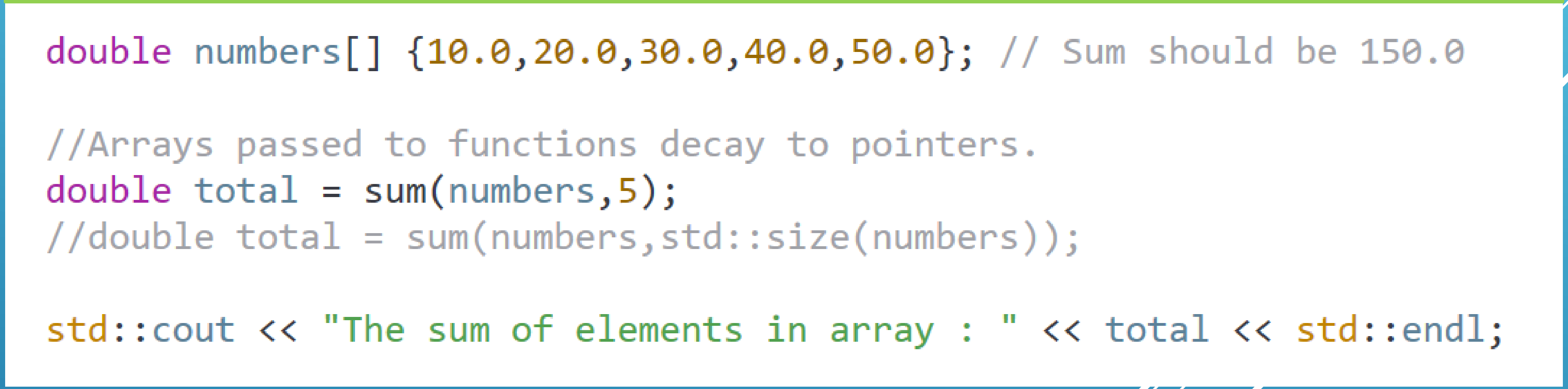
Arrays as function parameters

Array function parameters



```cpp
double sum ( double array[], size_t count){
//double sum ( double * array, size_t count){
    double sum{};
    //array = &sum;// This compiles!

    array[1] = 70.0; // Can affect the real arg array in the function taking array arg

    std::cout << "size of array : " << sizeof(array) << std::endl;
    std::cout << "size of int* : " << sizeof(int *) << std::endl;
    std::cout << "size of long int* : " << sizeof(long int *) << std::endl;

    //size_t size {std::size(array)}; //Compiler error

    for(size_t i{} ; i < count ; ++i){
        //sum += *(array + i) ;
        //sum += array[i];
        sum += *(array++);// Can remove the parantheses here, but they make things more readable.
    }
    return sum;
}
```

Call the function



```
double numbers[] {10.0,20.0,30.0,40.0,50.0}; // Sum should be 150.0

//Arrays passed to functions decay to pointers.
double total = sum(numbers,5);
//double total = sum(numbers,std::size(numbers));

std::cout << "The sum of elements in array : " << total << std::endl;
```

Summary

- The array syntax is treated by the compiler as if you had passed just by pointer, the two shown syntaxes are equivalent.

- The array variable name is passed by value as a pointer. We are working on a copy in the function, so we can use it to manipulate array data. We can even change where the array argument pointer points (which we couldn’t do for a real array)

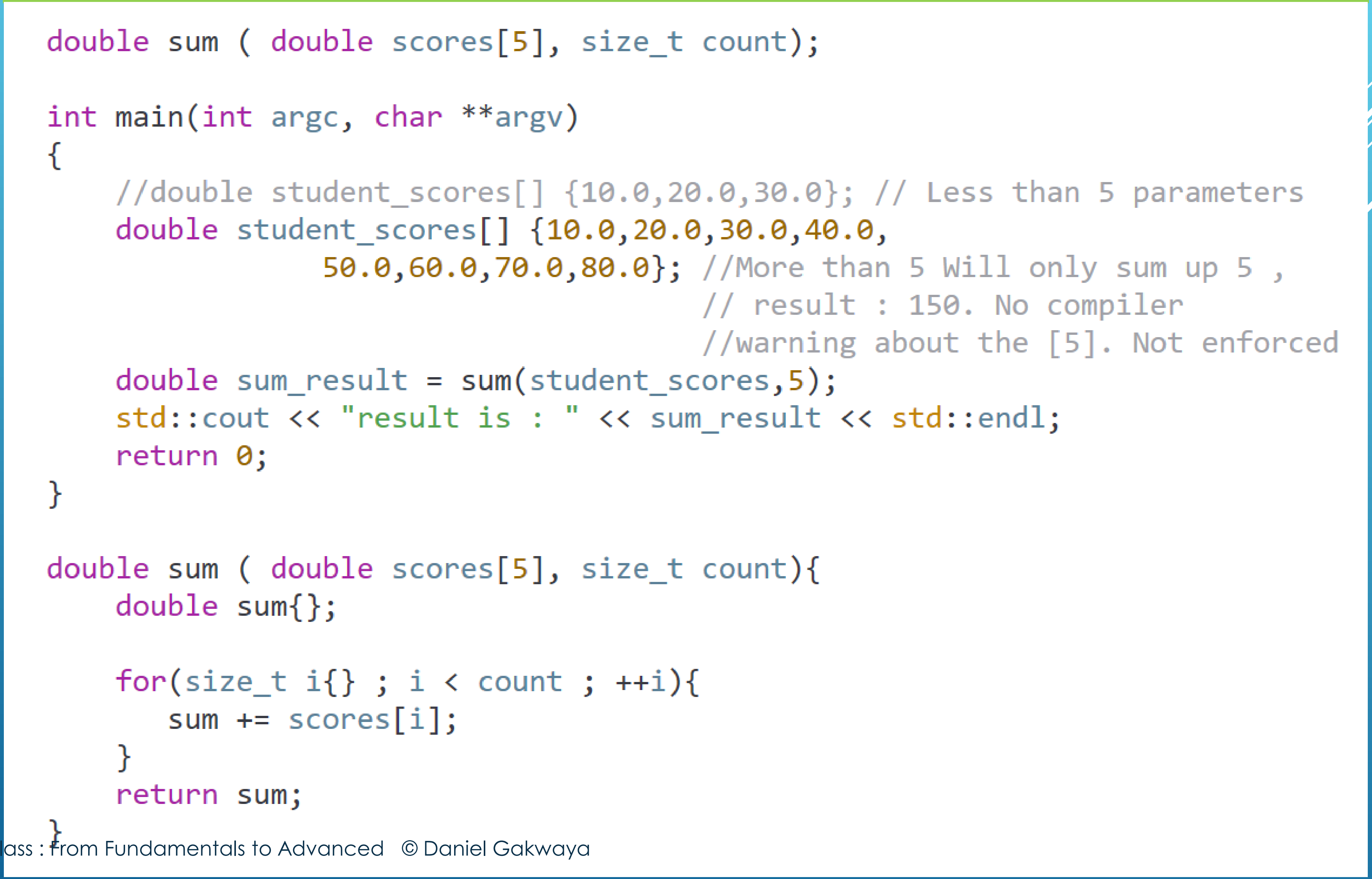
- There are limitations though : since the array has “decayed” into a pointer, we have lost all information related to the size of the array.For example sizeof will just return the size of the pointer, and std::size() won’t work at all inside the function

- When passing arrays as function parameters, we usually pass the size of the array as a second parameter. We have no way of getting that information inside the function

Slide intentionally left empty

Sized Array Function Parameters

Sized array function parameters

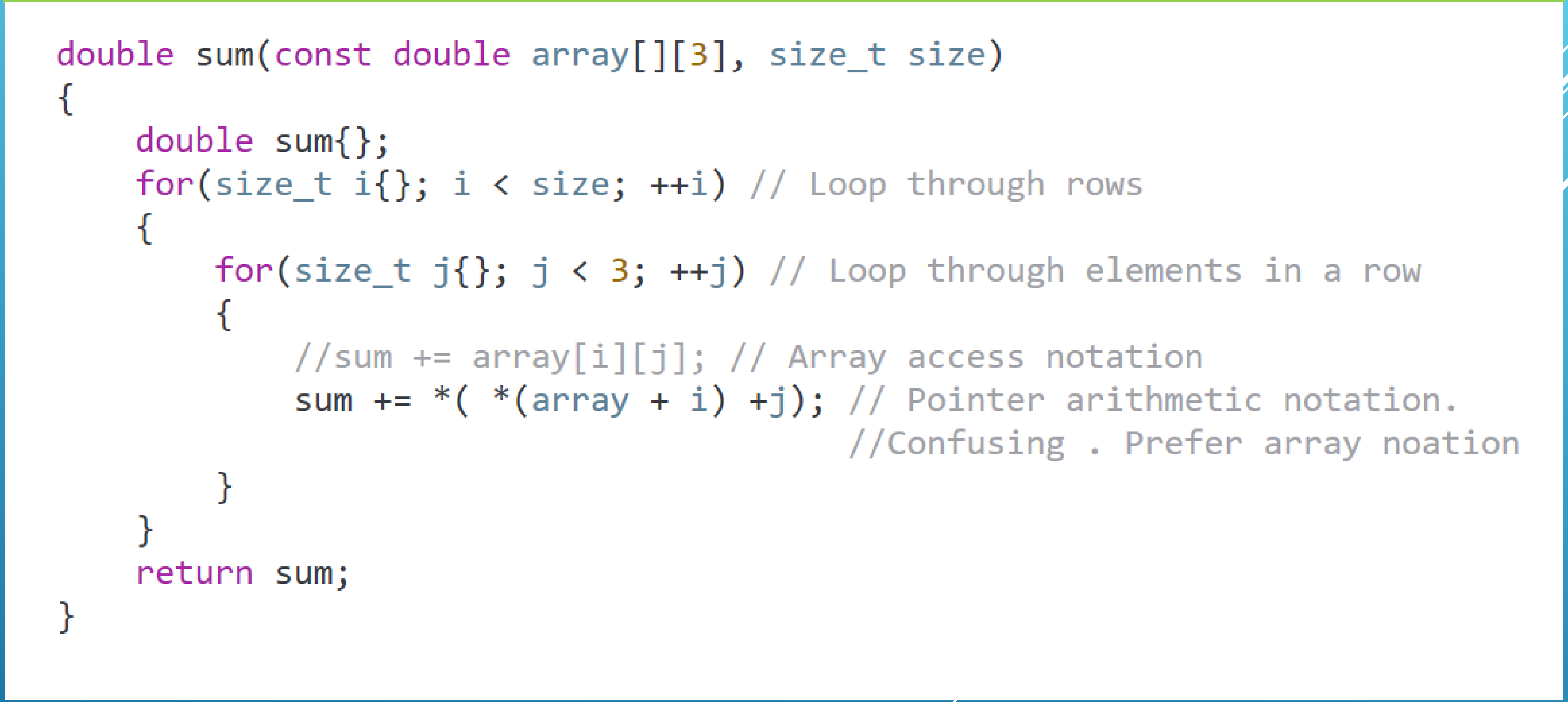


```cpp
double sum ( double scores[5], size_t count);

int main(int argc, char **argv)
{
    //double student_scores[] {10.0,20.0,30.0}; // Less than 5 parameters
    double student_scores[] {10.0,20.0,30.0,40.0,
                  50.0,60.0,70.0,80.0}; //More than 5 Will only sum up 5 ,
                                      // result : 150. No compiler
                                      //warning about the [5]. Not enforced
    double sum_result = sum(student_scores,5);
    std::cout << "result is : " << sum_result << std::endl;
    return 0;
}

double sum ( double scores[5], size_t count){
    double sum{};

    for(size_t i{} ; i < count ; ++i){
        sum += scores[i];
    }
    return sum;
}
```

Slide intentionally left empty

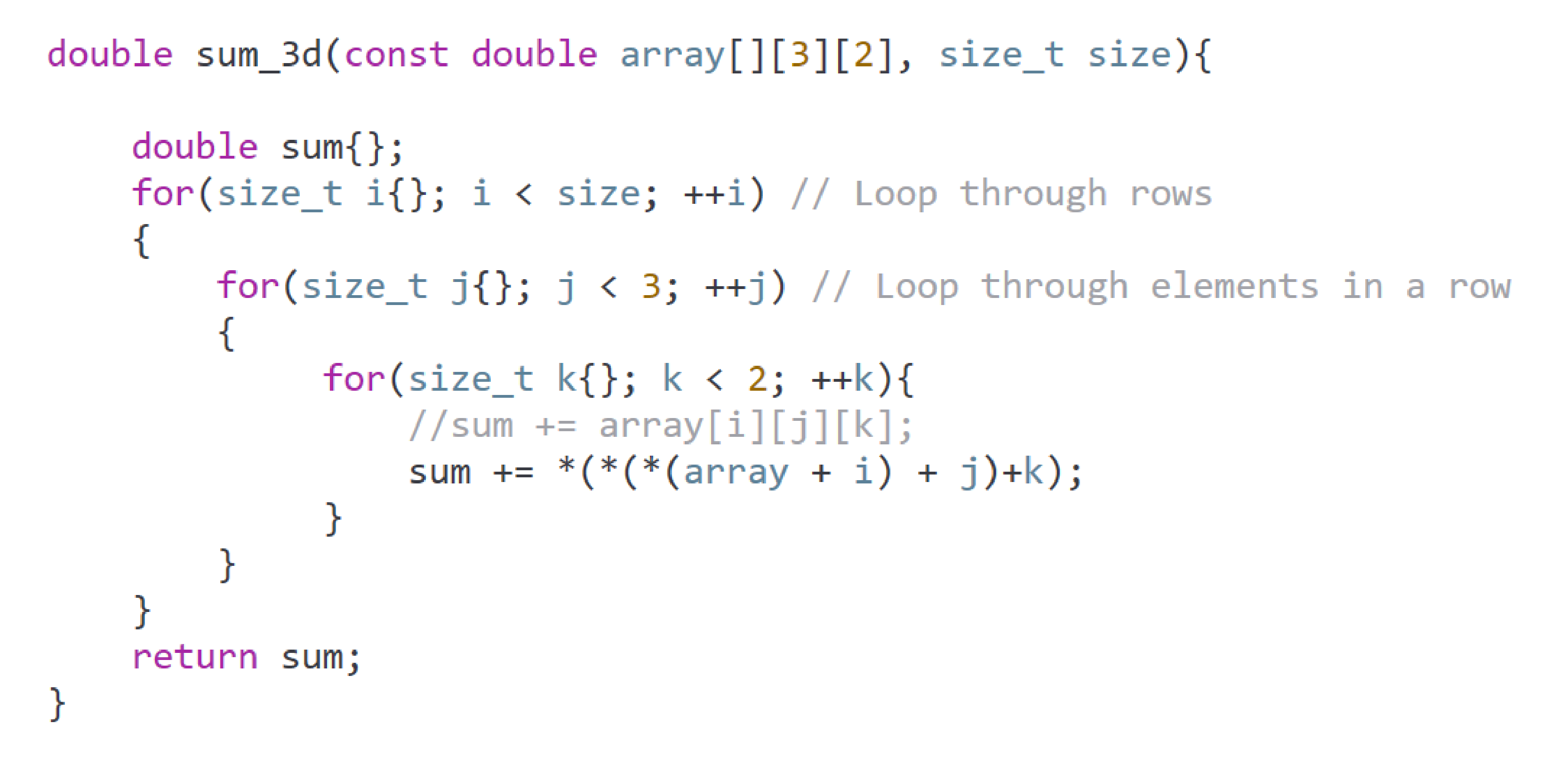
Multi dimensional array function parameters

2d array



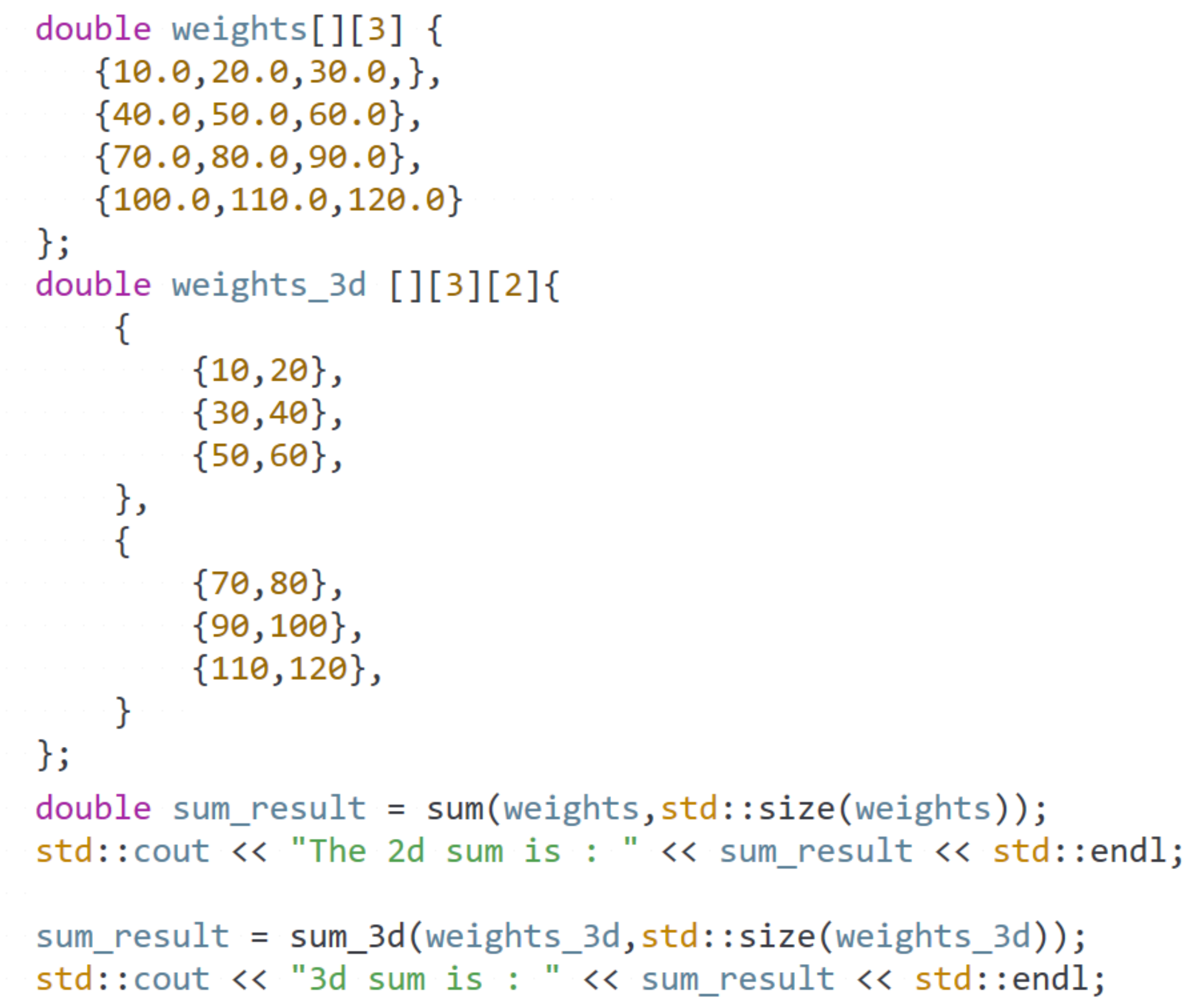
```
double sum(const double array[][3], size_t size)
{
    double sum{};
    for(size_t i{}; i < size; ++i) // Loop through rows
    {
        for(size_t j{}; j < 3; ++j) // Loop through elements in a row
        {
            //sum += array[i][j]; // Array access notation
            sum += *( *(array + i) +j); // Pointer arithmetic notation.
                                        //Confusing . Prefer array noation
        }
    }
    return sum;
}
```

3d array



```
double sum_3d(const double array[][3][2], size_t size){

    double sum{};
    for(size_t i{}; i < size; ++i) // Loop through rows
    {
        for(size_t j{}; j < 3; ++j) // Loop through elements in a row
        {
            for(size_t k{}; k < 2; ++k){
                //sum += array[i][j][k];
                sum += *(*(*(array + i) + j)+k);
            }
        }
    }
    return sum;
}
```

3d array

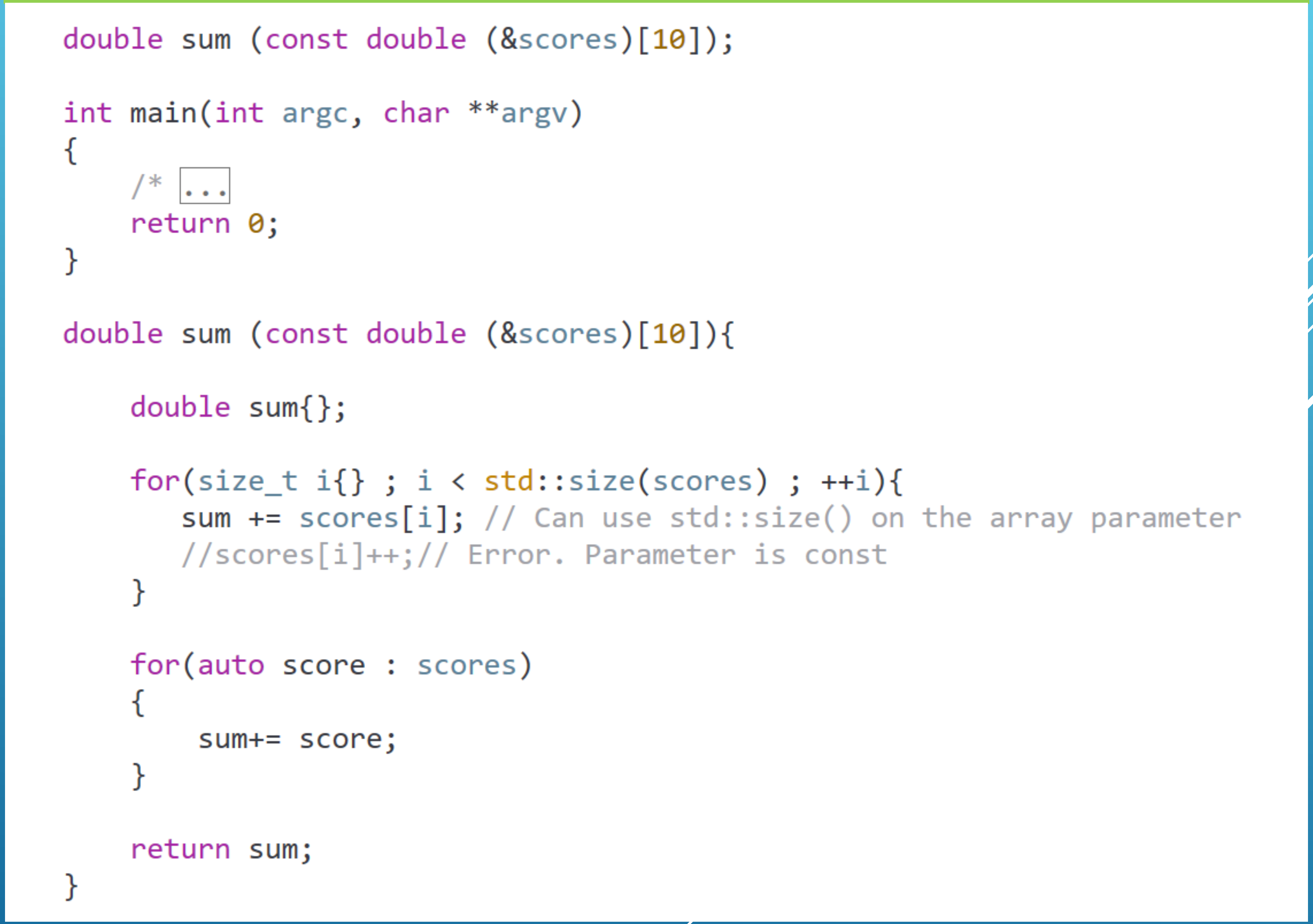


```
double weights[][3] {
   {10.0,20.0,30.0,},
   {40.0,50.0,60.0},
   {70.0,80.0,90.0},
   {100.0,110.0,120.0}
};
double weights_3d [][3][2]{
    {
        {10,20},
        {30,40},
        {50,60},
    },
    {
        {70,80},
        {90,100},
        {110,120},
    }
};
double sum_result = sum(weights,std::size(weights));
std::cout << "The 2d sum is : " << sum_result << std::endl;

sum_result = sum_3d(weights_3d,std::size(weights_3d));
std::cout << "3d sum is : " << sum_result << std::endl;
```

Slide intentionally left empty

Sized array function parameters by reference

Sized array passed by reference

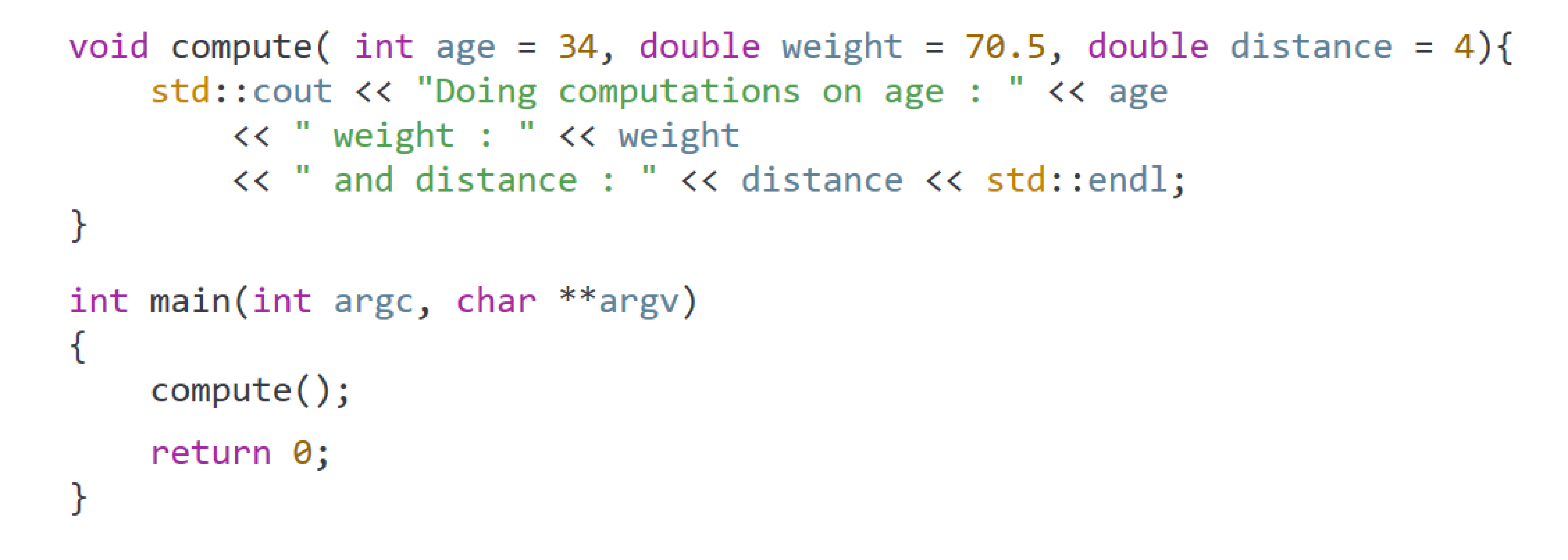


```
double sum (const double (&scores)[10]);

int main(int argc, char **argv)
{
    /* ...
    return 0;
}

double sum (const double (&scores)[10]){

    double sum{};

    for(size_t i{} ; i < std::size(scores) ; ++i){
       sum += scores[i]; // Can use std::size() on the array parameter
       //scores[i]++;// Error. Parameter is const
    }

    for(auto score : scores)
    {
        sum+= score;
    }

    return sum;
}
```

Slide intentionally left empty

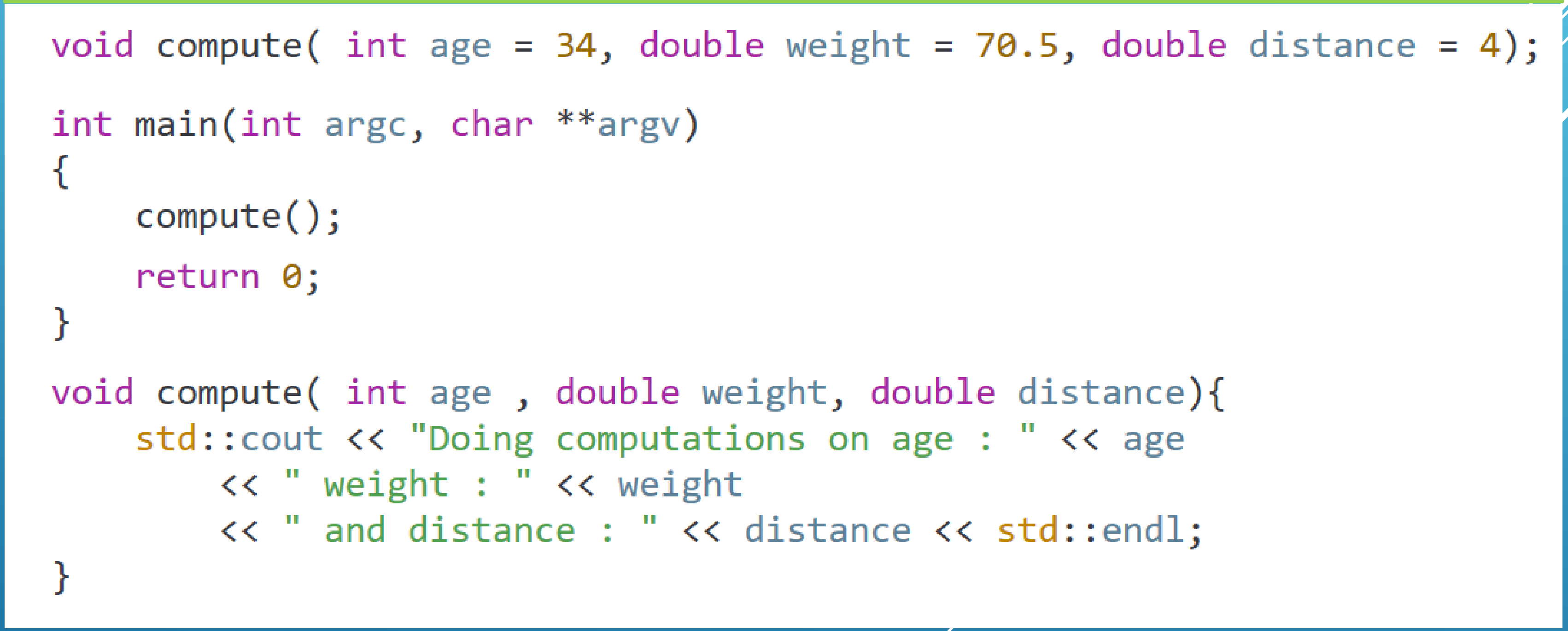
Default function arguments

Default arguments : syntax



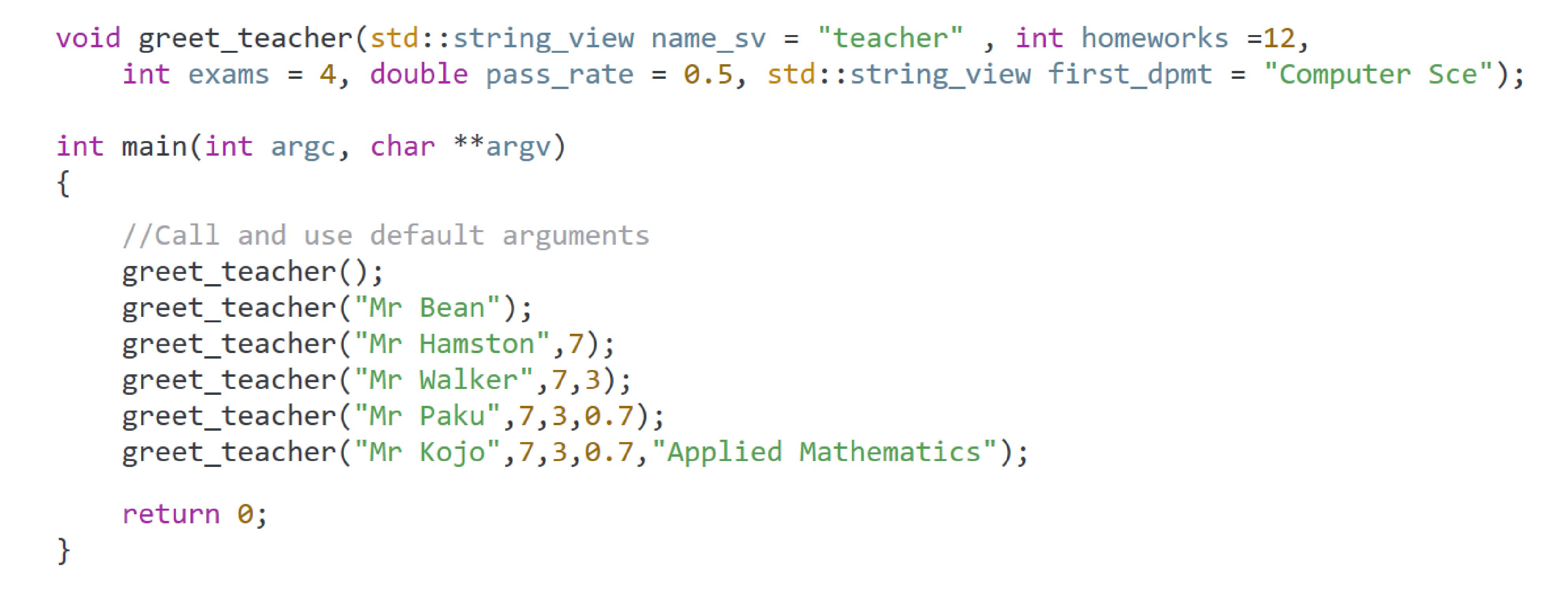
```cpp
void compute( int age = 34, double weight = 70.5, double distance = 4){
    std::cout << "Doing computations on age : " << age
        << " weight : " << weight
        << " and distance : " << distance << std::endl;
}

int main(int argc, char **argv)
{
    compute();

    return 0;
}
```

Declaration - Definition : syntax



```cpp
void compute( int age = 34, double weight = 70.5, double distance = 4);

int main(int argc, char **argv)
{
    compute();

    return 0;
}

void compute( int age , double weight, double distance){
    std::cout << "Doing computations on age : " << age
        << " weight : " << weight
        << " and distance : " << distance << std::endl;
}
```

Just another example

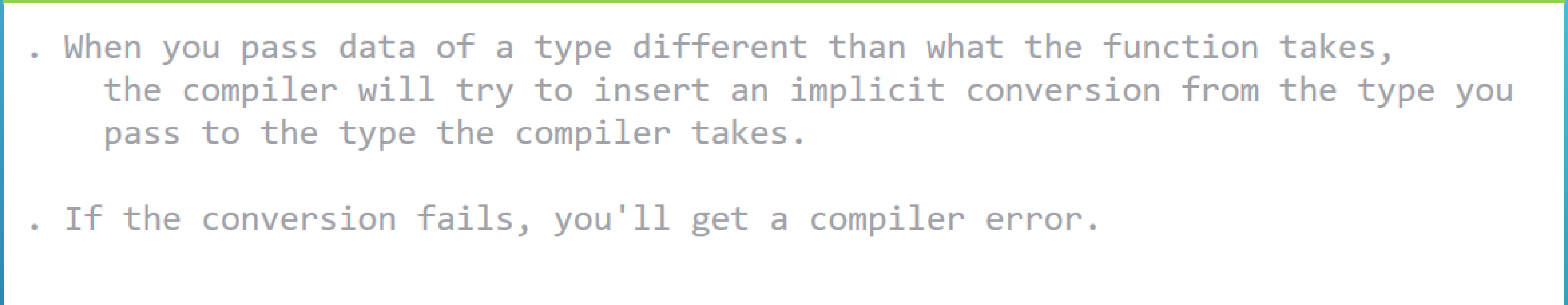


```
void greet_teacher(std::string_view name_sv = "teacher" , int homeworks =12,
    int exams = 4, double pass_rate = 0.5, std::string_view first_dpmt = "Computer Sce");

int main(int argc, char **argv)
{
    //Call and use default arguments
    greet_teacher();
    greet_teacher("Mr Bean");
    greet_teacher("Mr Hamston",7);
    greet_teacher("Mr Walker",7,3);
    greet_teacher("Mr Paku",7,3,0.7);
    greet_teacher("Mr Kojo",7,3,0.7,"Applied Mathematics");

    return 0;
}
```

Slide intentionally left empty

Implicit Conversions

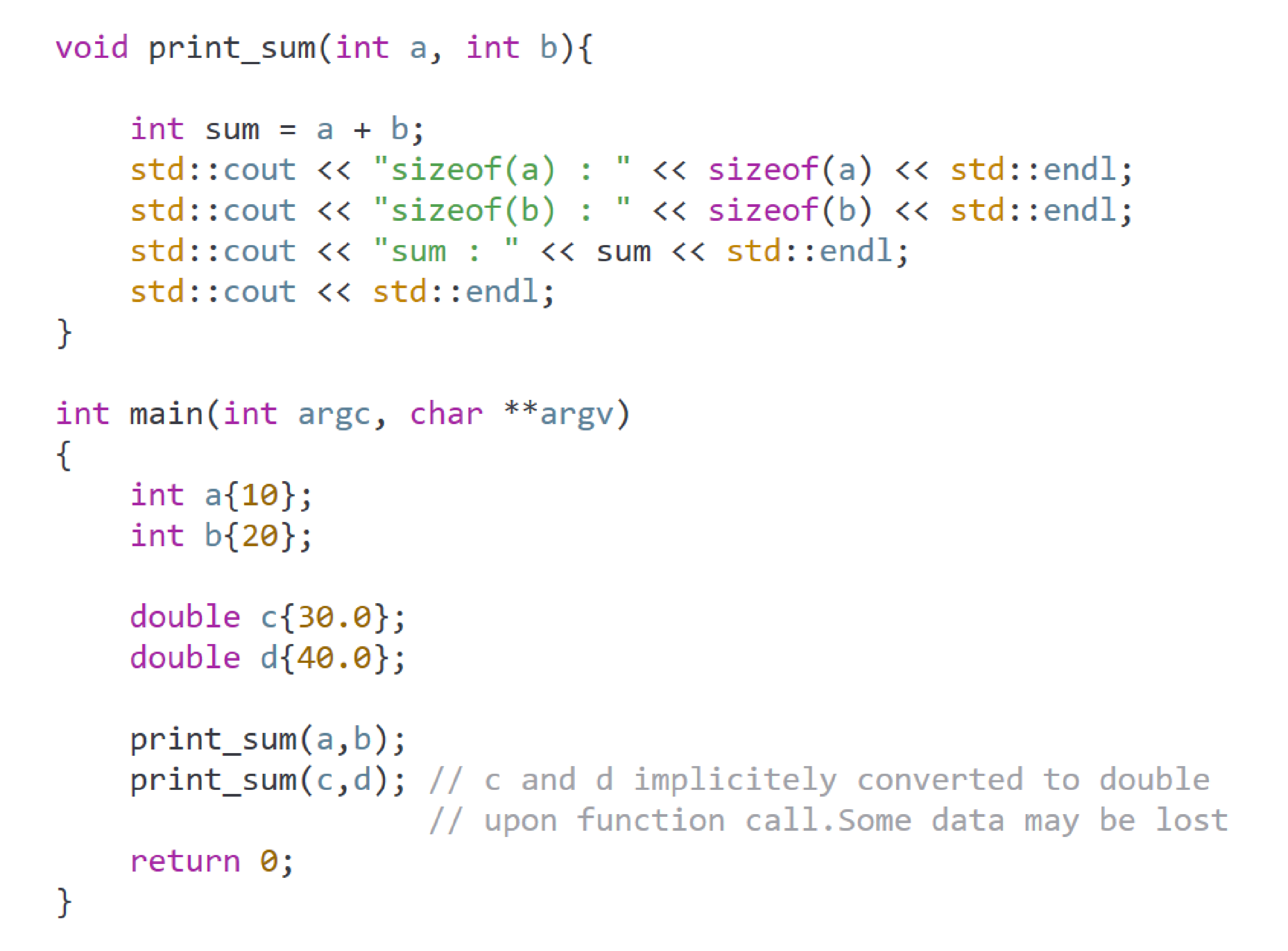
Implicit conversions



. When you pass data of a type different than what the function takes, the compiler will try to insert an implicit conversion from the type you pass to the type the compiler takes.

. If the conversion fails, you'll get a compiler error.

Implicit conversions

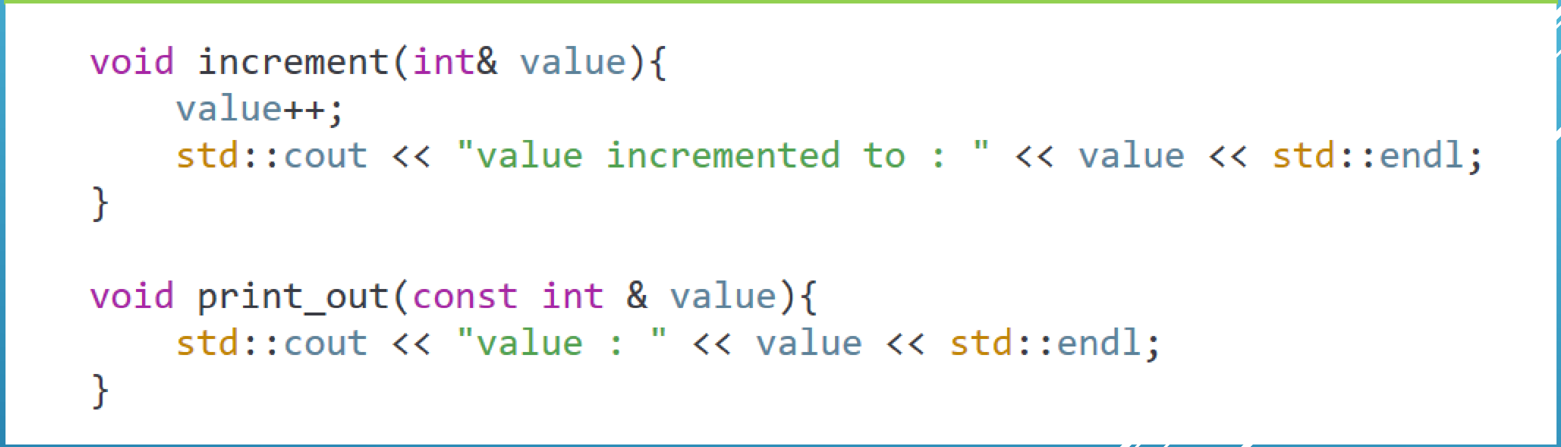


```
void print_sum(int a, int b){

    int sum = a + b;
    std::cout << "sizeof(a) : " << sizeof(a) << std::endl;
    std::cout << "sizeof(b) : " << sizeof(b) << std::endl;
    std::cout << "sum : " << sum << std::endl;
    std::cout << std::endl;
}

int main(int argc, char **argv)
{
    int a{10};
    int b{20};

    double c{30.0};
    double d{40.0};

    print_sum(a,b);
    print_sum(c,d); // c and d implicitely converted to double
                    // upon function call.Some data may be lost
    return 0;
}
```

Slide intentionally left empty

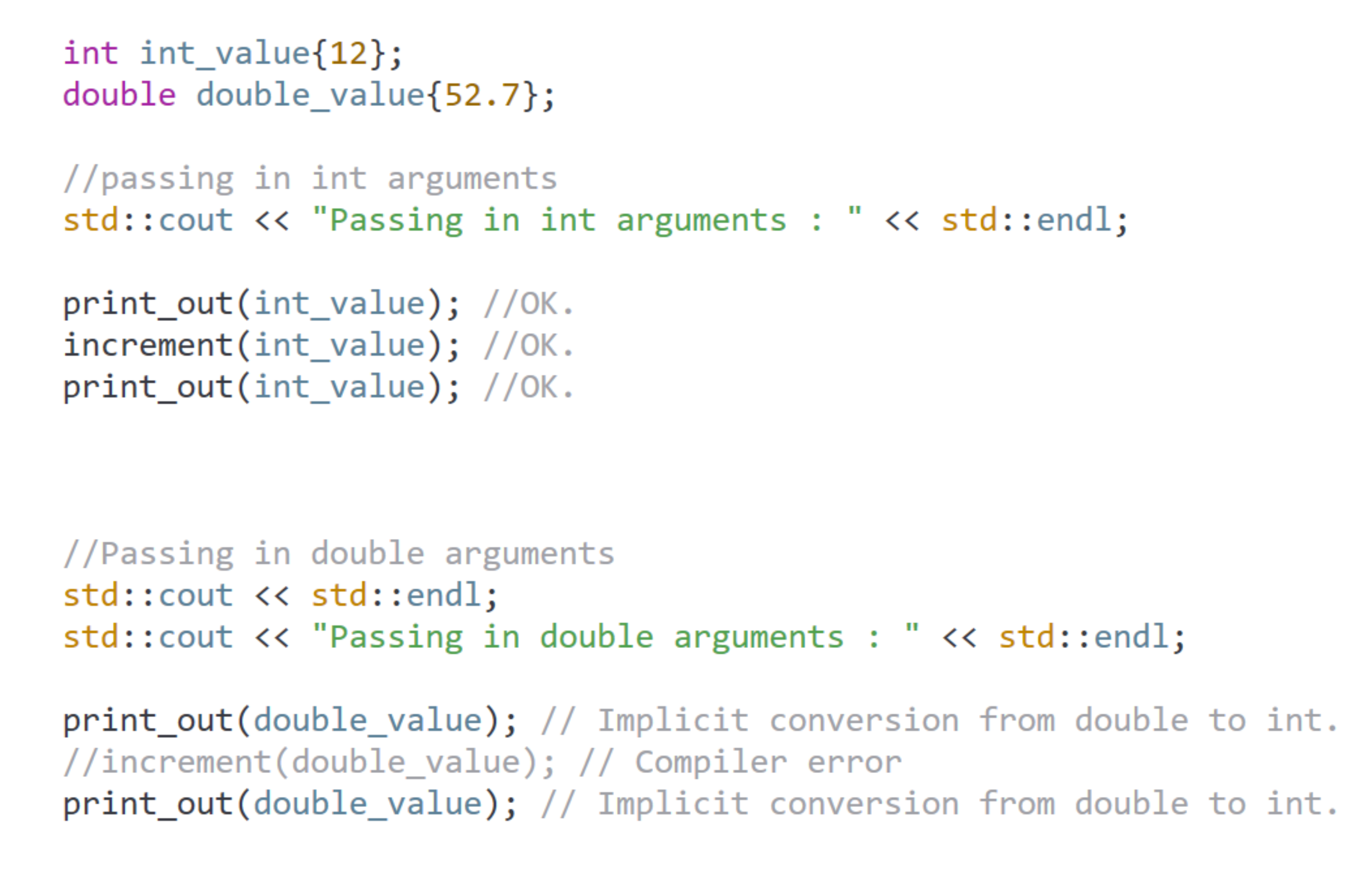
Implicit Conversions with references

Functions taking in references



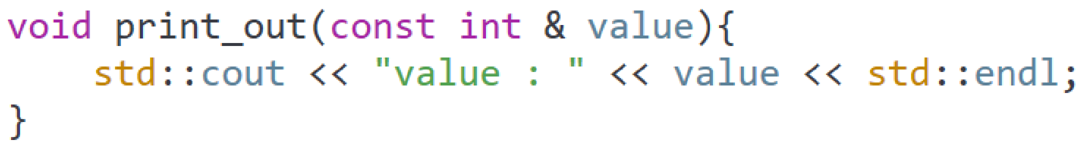
```
void increment(int& value){
    value++;
    std::cout << "value incremented to : " << value << std::endl;
}

void print_out(const int & value){
    std::cout << "value : " << value << std::endl;
}
```

Weird non const references!



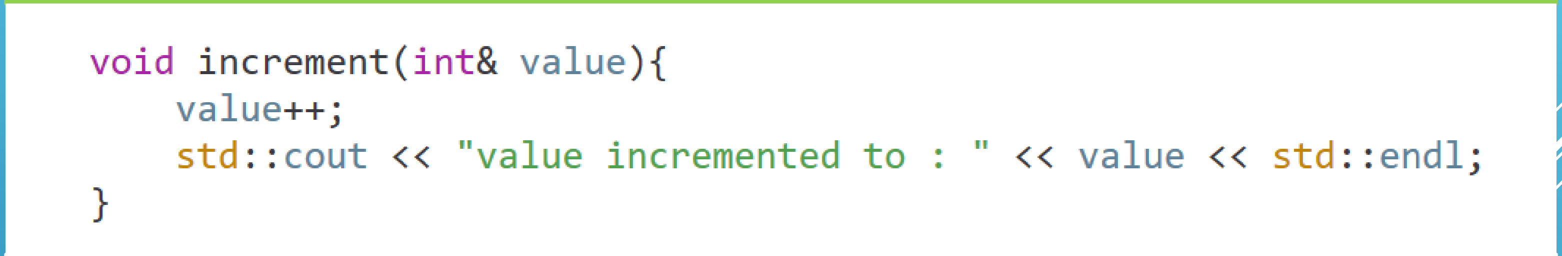
```
int int_value{12};
double double_value{52.7};

//passing in int arguments
std::cout << "Passing in int arguments : " << std::endl;

print_out(int_value); //OK.
increment(int_value); //OK.
print_out(int_value); //OK.



//Passing in double arguments
std::cout << std::endl;
std::cout << "Passing in double arguments : " << std::endl;

print_out(double_value); // Implicit conversion from double to int.
//increment(double_value); // Compiler error
print_out(double_value); // Implicit conversion from double to int.
```

Functions taking in const references



```
void print_out(const int & value){
    std::cout << "value : " << value << std::endl;
}
```

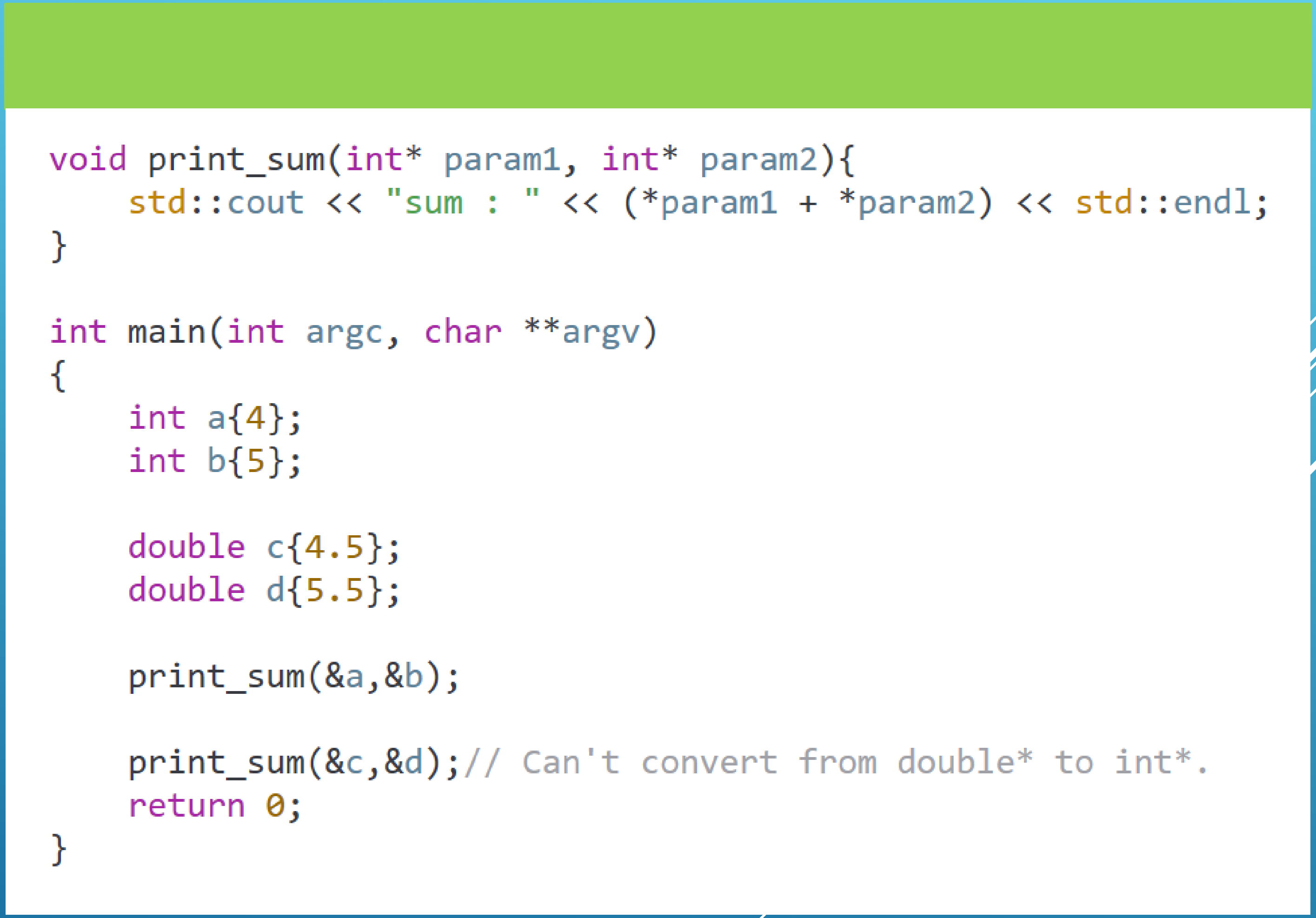
Functions taking in non const references



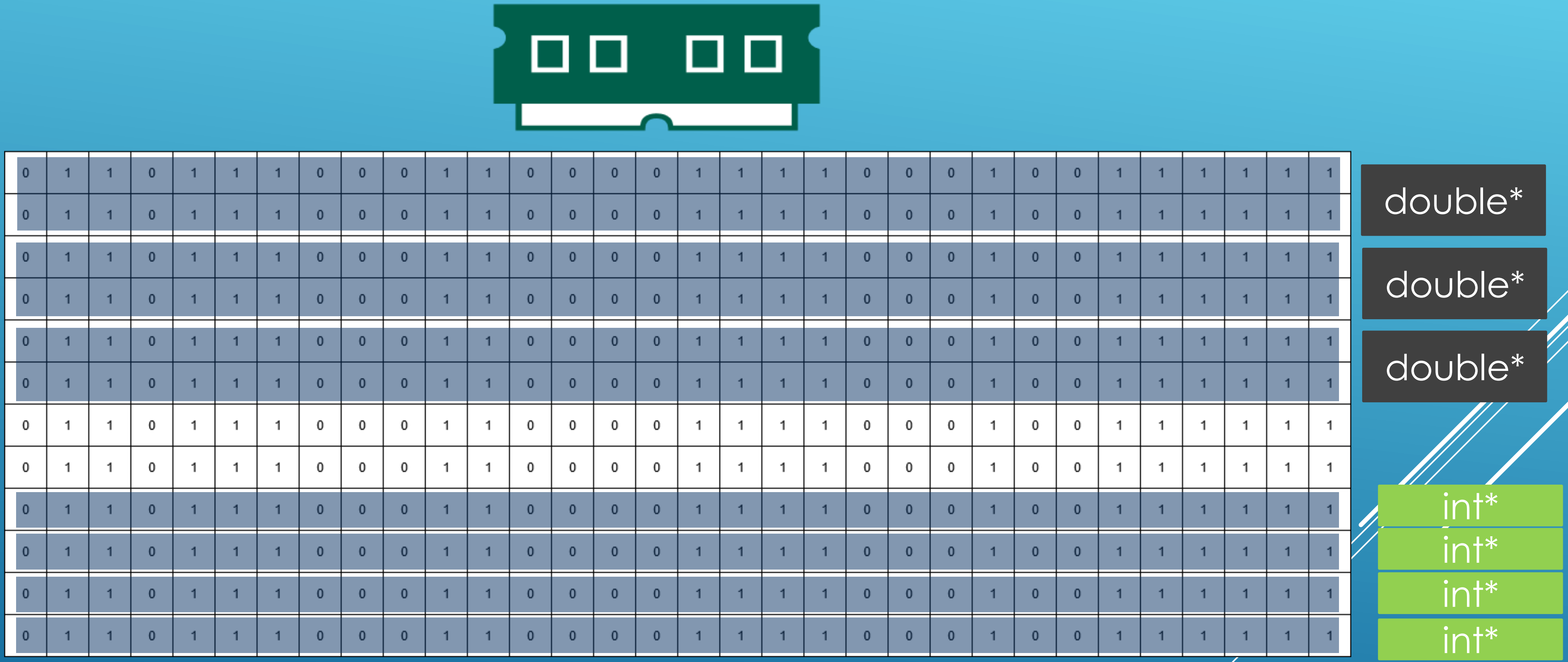
```
void increment(int& value){
    value++;
    std::cout << "value incremented to : " << value << std::endl;
}
```

Slide intentionally left empty

Implicit conversions with pointers



```cpp
void print_sum(int* param1, int* param2){
    std::cout << "sum : " << (*param1 + *param2) << std::endl;
}

int main(int argc, char **argv)
{
    int a{4};
    int b{5};

    double c{4.5};
    double d{5.5};

    print_sum(&a,&b);

    print_sum(&c,&d);// Can't convert from double* to int*.
    return 0;
}
```

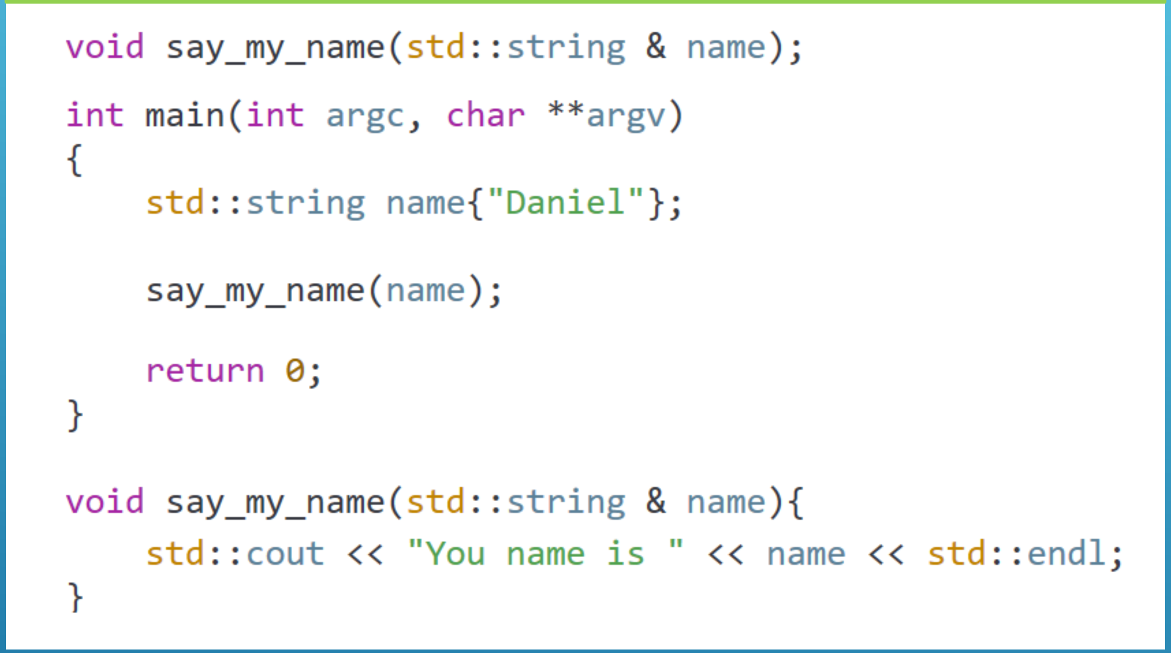


0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1

Slide intentionally left empty

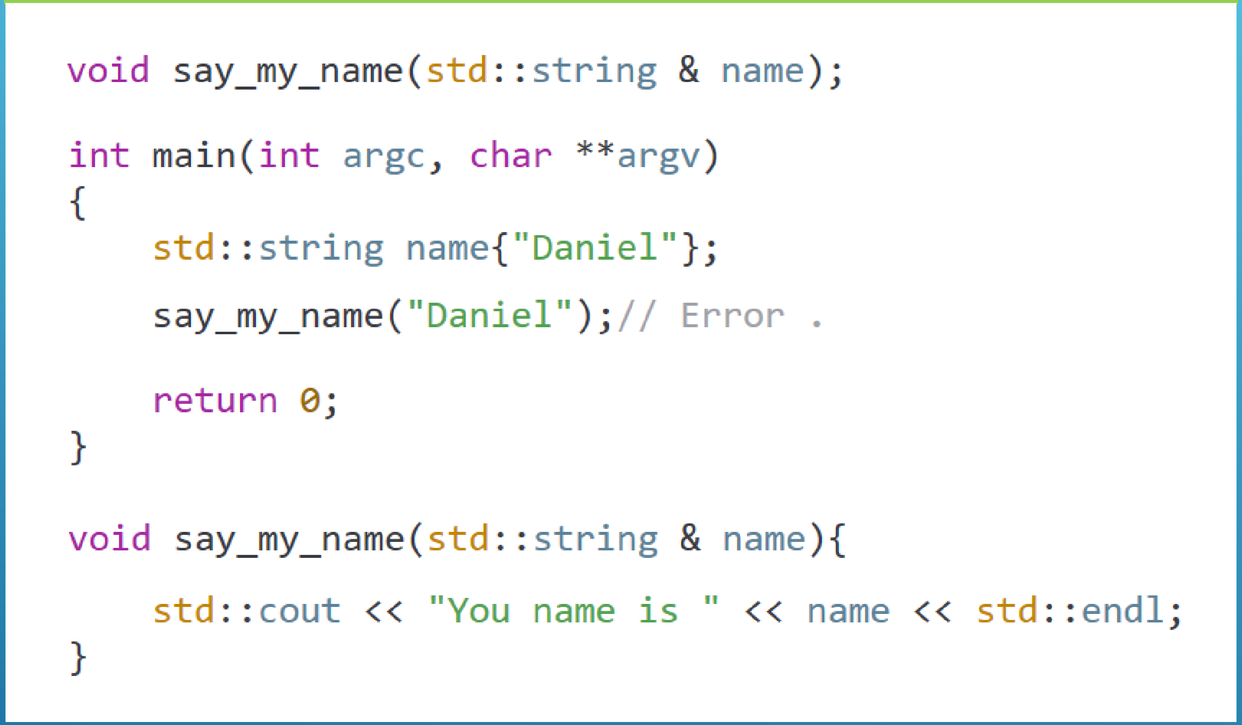
Stringview Parameters

Passing in a predefined std::string



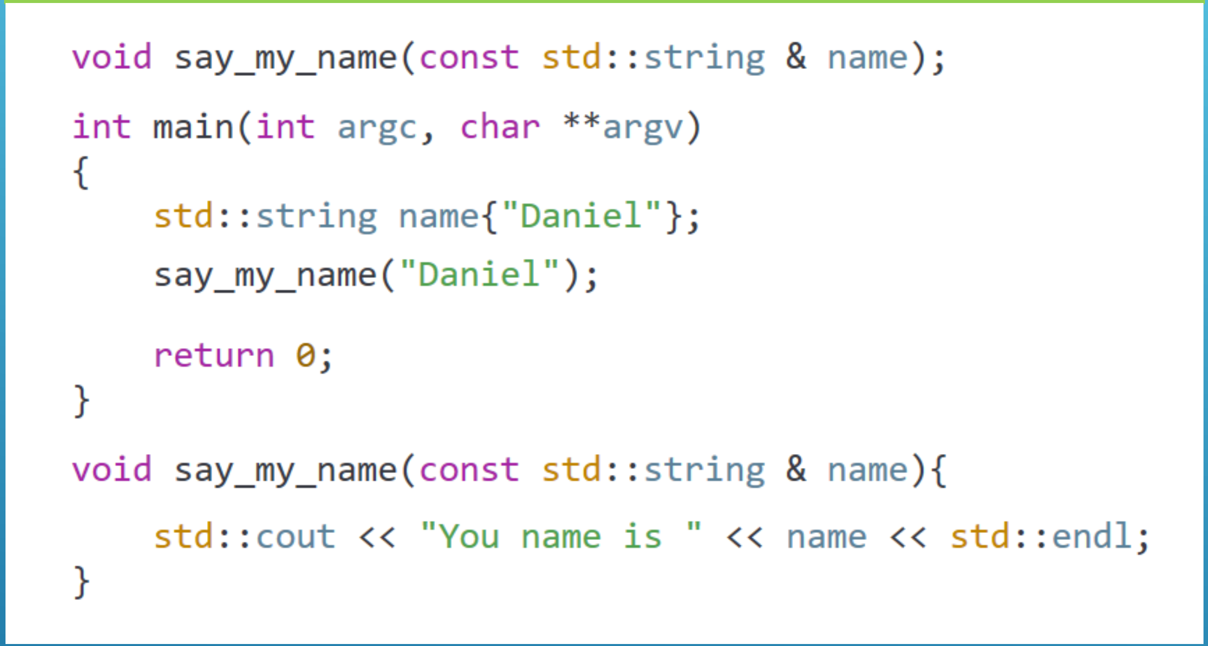
```cpp
void say_my_name(std::string & name);

int main(int argc, char **argv)
{
    std::string name{"Daniel"};

    say_my_name(name);

    return 0;
}

void say_my_name(std::string & name){
    std::cout << "You name is " << name << std::endl;
}
```

Passing in a string literal. BAD!



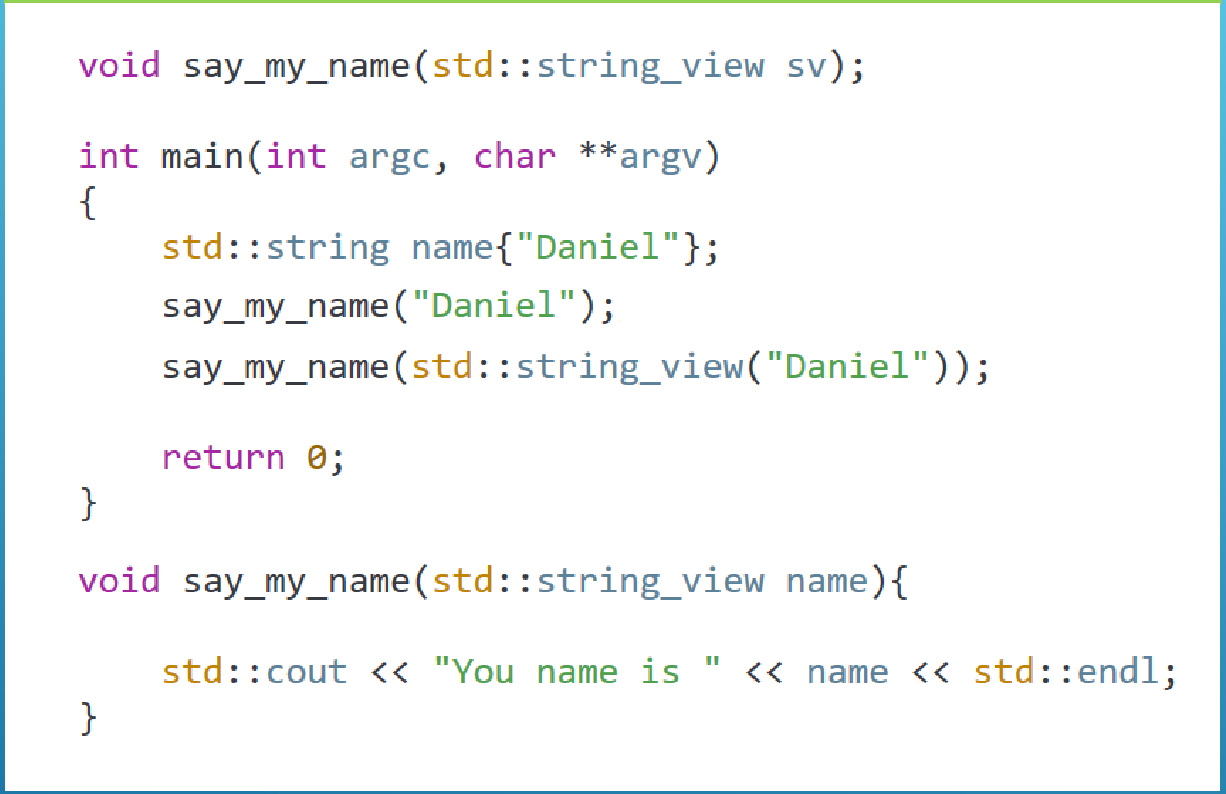
```
void say_my_name(std::string & name);

int main(int argc, char **argv)
{
    std::string name{"Daniel"};

    say_my_name("Daniel");// Error .

    return 0;
}

void say_my_name(std::string & name){

    std::cout << "You name is " << name << std::endl;
}
```

Function takes const std::string&



```
void say_my_name(const std::string & name);

int main(int argc, char **argv)
{
    std::string name{"Daniel"};
    say_my_name("Daniel");

    return 0;
}

void say_my_name(const std::string & name){
    std::cout << "You name is " << name << std::endl;
}
```

Function takes std::string_view



```
void say_my_name(std::string_view sv);

int main(int argc, char **argv)
{
    std::string name{"Daniel"};
    say_my_name("Daniel");
    say_my_name(std::string_view("Daniel"));

    return 0;
}

void say_my_name(std::string_view name){

    std::cout << "You name is " << name << std::endl;
}
```

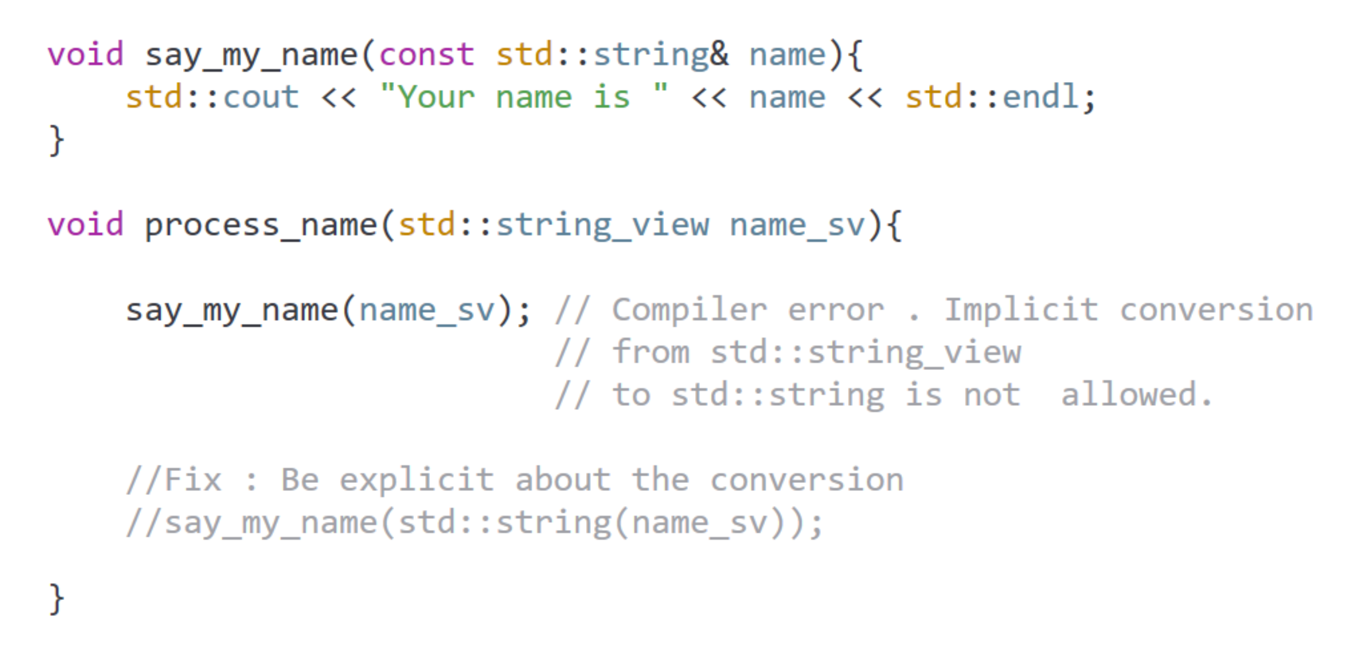
General recomendation

Where possible, always use std::string_view for string input in functions, and const references for other types

Slide intentionally left empty

Implicit Conversions from std::string_view to std::string

std::string_view to std::string



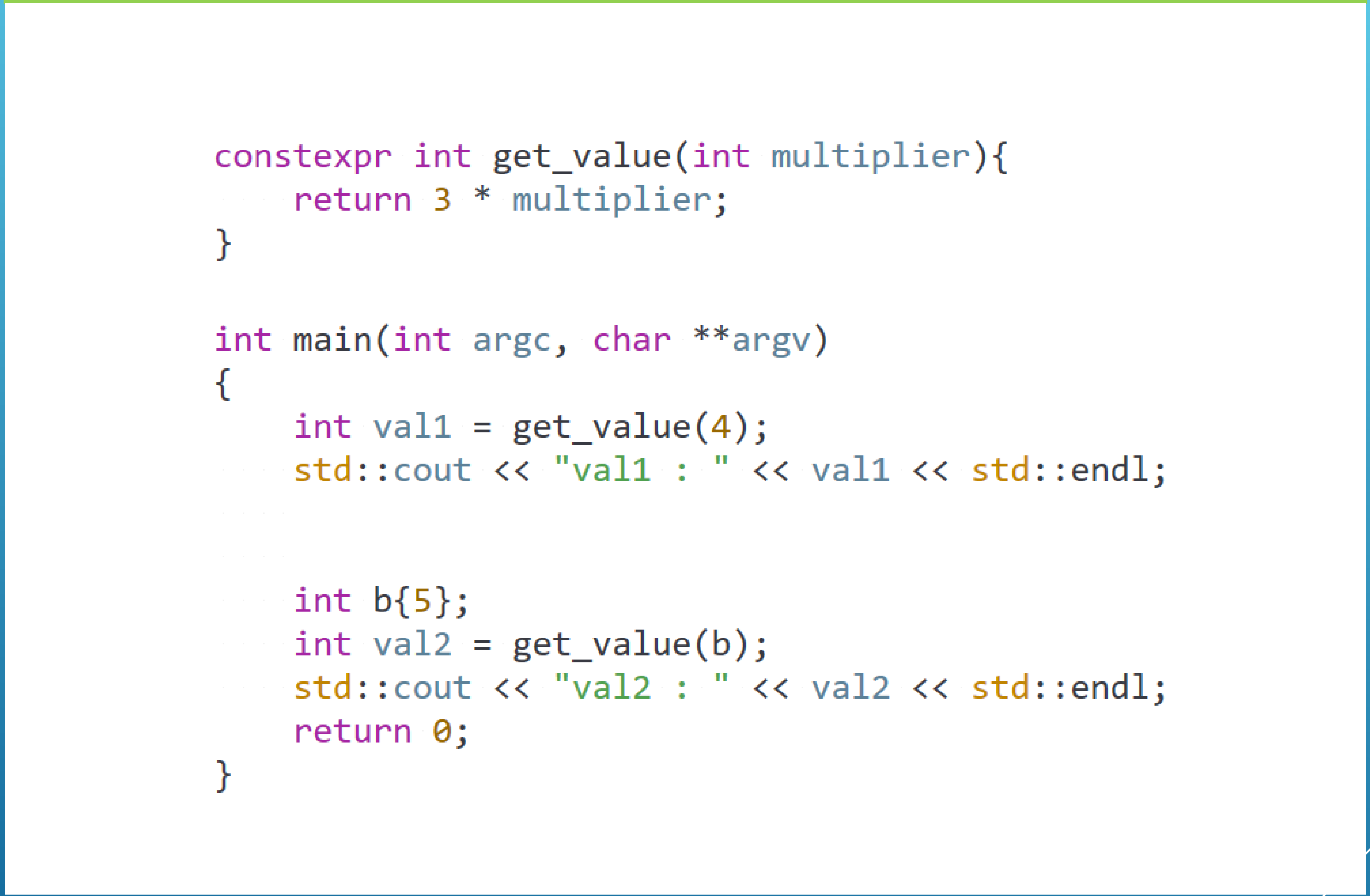
```cpp
void say_my_name(const std::string& name){
    std::cout << "Your name is " << name << std::endl;
}

void process_name(std::string_view name_sv){

    say_my_name(name_sv); // Compiler error . Implicit conversion
                          // from std::string_view
                          // to std::string is not  allowed.

    //Fix : Be explicit about the conversion
    //say_my_name(std::string(name_sv));

}
```

Slide intentionally left empty

Constexpr Functions

Compile time

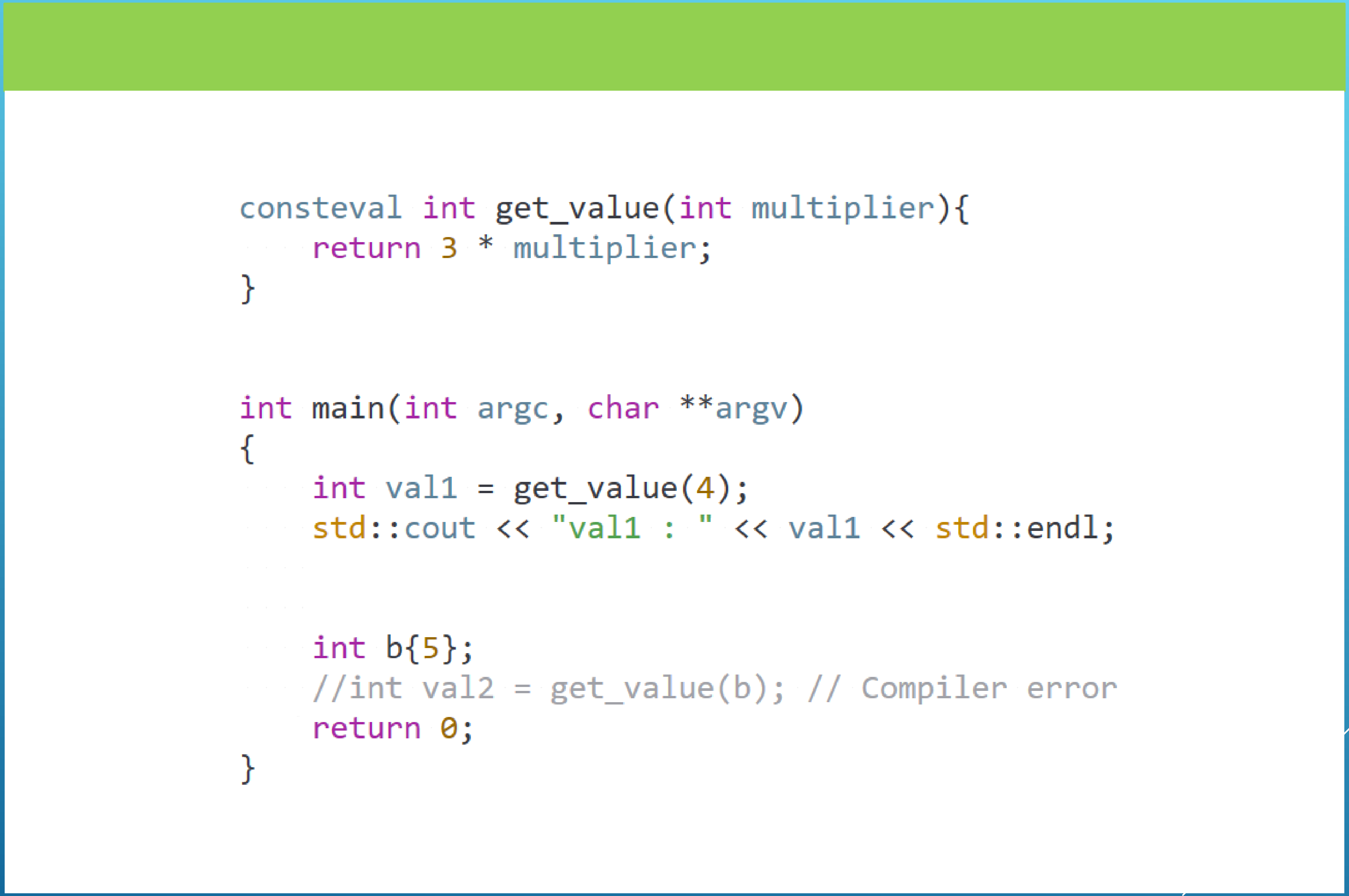
Run time



```
constexpr int get_value(int multiplier){
    return 3 * multiplier;
}

int main(int argc, char **argv)
{
    int val1 = get_value(4);
    std::cout << "val1 : " << val1 << std::endl;


    int b{5};
    int val2 = get_value(b);
    std::cout << "val2 : " << val2 << std::endl;
    return 0;
}
```

Slide intentionally left empty

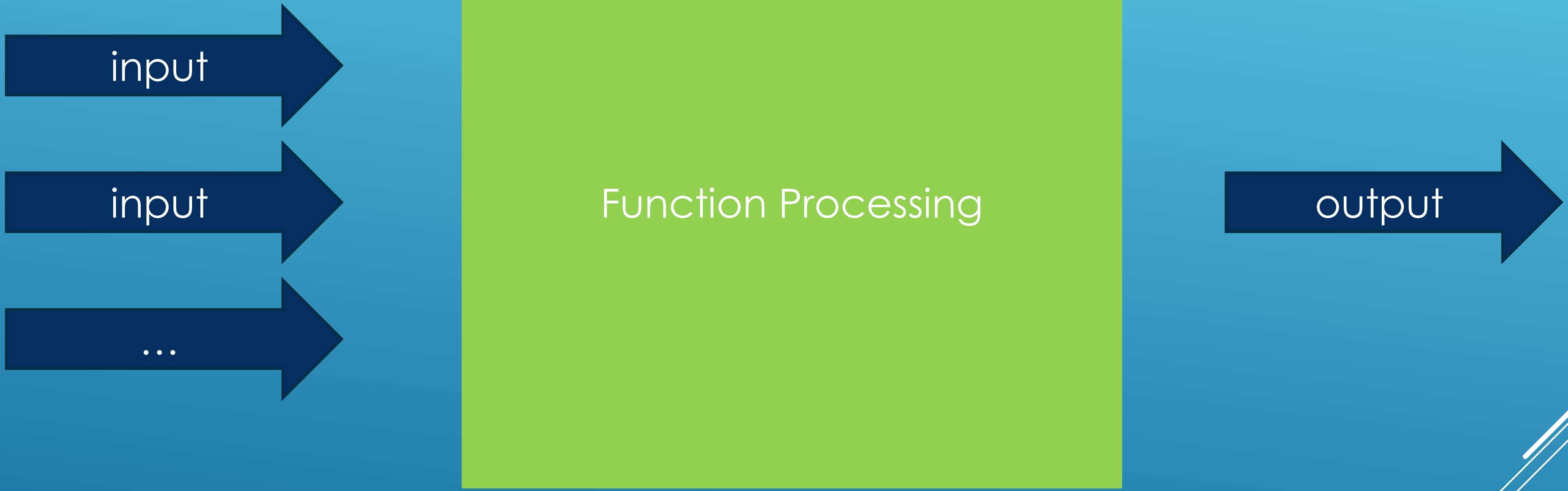
consteval Functions



```cpp
consteval int get_value(int multiplier){
    return 3 * multiplier;
}


int main(int argc, char **argv)
{
    int val1 = get_value(4);
    std::cout << "val1 : " << val1 << std::endl;


    int b{5};
    //int val2 = get_value(b); // Compiler error
    return 0;
}
```

Slide intentionally left empty

Functions : Summary



input

input

...

Function Processing

output

Top down programming in the main function

Reusable code components we can call several times in the main function

Signature,prototype

Declaration and definition

conseval and constexpr functions

Multiple files

- Preprocessing
- Compilation
- Linking

Default arguments

- Pass by value
- Pass by reference
- Pass by pointer

Array function parameters

Implicit conversions

Slide intentionally left empty