

Modern C++ for Computer Vision and Image Processing

Lecture 3: C++ Functions

Ignacio Vizzo and Cyrill Stachniss

C-style strings are evil

Like everything else in C in general.

```
1 #include <cstring>
2 #include <iostream>
3
4 int main() {
5     const char source[] = "Copy this!";
6     char dest[5];
7     std::cout << source << '\n';
8
9     std::strcpy(dest, source);
10    std::cout << dest << '\n';
11
12    // source is const, no problem right?
13    std::cout << source << '\n';
14
15    return 0;
16 }
```

Strings

- `#include <string>` to use `std::string`
- Concatenate strings with `+`
- Check if `str` is empty with `str.empty()`
- Works out of the box with I/O streams

```
1 #include <iostream>
2 #include <string>
3
4 int main() {
5     const std::string source{"Copy this!"};
6     std::string dest = source; // copy string
7
8     std::cout << source << '\n';
9     std::cout << dest << '\n';
10    return 0;
11 }
```

Function definition

*In programming, a named section of a program that performs a **specific** task. In this sense, a **function** is a type of **procedure** or **routine**. Some programming languages make a distinction between a **function**, which returns a value, and a **procedure**, which performs some operation but does not return a value.*

Bjarne Stroustrup

*The main way of getting something done in a C++ program is to call a **function** to do it. Defining a **function** is the way you specify how an operation is to be done. A **function** cannot be called unless it has been previously declared. A **function** declaration gives the name of the **function**, the type of the value returned (if any), and the number and types of the arguments that must be supplied in a call.*

Functions

```
1 ReturnType FuncName(ParamType1 in_1, ParamType2 in_2) {  
2     // Some awesome code here.  
3     return return_value;  
4 }
```

- Code can be organized into functions
- Functions **create a scope**
- **Single return value** from a function
- Any number of input variables of any types
- Should do **only one** thing and do it right
- Name **must** show what the function does
- **GOOGLE-STYLE** name functions in **CamelCase**
- **GOOGLE-STYLE** **write small functions**

Function Anatomy

```
1 [[attributes]] ReturnType FuncName(ArgumentList...) {  
2     // Some awesome code here.  
3     return return_value;  
4 }
```

- Body
- Optional Attributes
- Return Type
- Name
- Argument List

Function Body

- Where the **computation** happens.
- Defines a new scope, the `scope of the function`.
- Access outside world(scopes) through input arguments.
- Can not add information about the implementation outside this `scope`

Funtcion Body

```
1 // This is not part of the body of the function
2
3 void MyFunction() {
4     // This is the body of the function
5     // Whatever is inside here is part of
6     // the scope of the function
7 }
8
9 // This is not part of the body of the function
```

Return Type

Could be any of:

1. An **unique** type, eg: `int`, `std::string`, etc...
2. `void`, also called subroutine.

Rules:

- If has a return type, **must** return a value.
- If returns void, must **NOT** return any value.

Return Type

```
1 int f1() {}           // error
2 void f2() {}         // OK
3
4 int f3() { return 1; } // error
5 void f4() { return 1; } // OK
6
7 int f5() { return; } // error
8 void f6() { return; } // OK
```

Return Type

Automatic return type deduction (C++14):

```
1 std::map<char, int> GetDictionary() {  
2     return std::map<char, int>{{'a', 27}, {'b', 3}};  
3 }
```

Can be expressed as:

```
1 auto GetDictionary() {  
2     return std::map<char, int>{{'a', 27}, {'b', 3}};  
3 }
```

Return Type

Sadly you can use only one type for return values, so, no **Python** like:

```
1 #!/usr/bin/env python3
2 def foo():
3     return "Super Variable", 5
4
5
6 name, value = foo()
7 print(name + " has value: " + str(value))
```

Return Type

Sadly you can use only one type for return values, so no Python like:

```
1 #!/usr/bin/env python3
2 def foo():
3     return "Super Variable", 5
4
5
6 name, value = foo()
7 print(name + " has value: " + str(value))
```

Let's write this in C++, and make it run **18** times faster, with a similar syntax.

Return Type

With the introduction of structured binding in C++17 you can now:

```
1 #include <iostream>
2 #include <tuple>
3 using namespace std;
4
5 auto Foo() {
6     return make_tuple("Super Variable", 5);
7 }
8
9 int main() {
10     auto [name, value] = Foo();
11     cout << name << " has value :" << value << endl;
12     return 0;
13 }
```

Return Type

WARNING:

Never return reference to locally variables!!!

```
1 #include <iostream>
2 using namespace std;
3
4 int& MultiplyBy10(int num) { // retval is created
5     int retval = 0;
6     retval = 10 * num;
7     return retval;
8 } // retval is destroyed, it's not accesible anymore
9
10 int main() {
11     int out = MultiplyBy10(10);
12     cout << "out is " << out << endl;
13     return 0;
14 }
```


Return Type

```
1 #include <iostream>
2 using namespace std;
3
4 int& MultiplyBy10(int num) { // retval is created
5     int retval = 0;
6     retval = 10 * num;
7     cout << "retval is " << retval << endl;
8     return retval;
9 } // retval is destroyed, it's not accesible anymore
10
11 int main() {
12     int out = MultiplyBy10(10);
13     cout << "out is " << out << endl;
14     return 0;
15 }
```

Return Type

Compiler got your back:

Return value optimization:

https://en.wikipedia.org/wiki/Copy_elision#Return_value_optimization

```
1 Type DoSomething() {
2     Type huge_variable;
3
4     // ... do something
5
6     // don't worry, the compiler will optimize it
7     return huge_variable;
8 }
9
10 // ...
11
12 Type out = DoSomething(); // does not copy
```

Local Variables

- A local variable is initialized when the execution reaches its definition.

```
1 void f() {
2     // Gets initialized when the execution reaches
3     // the definition of f(), thus, this implementation
4     int local_variable = 50;
5 }
6
7 // at this point local_variable has not been initialized
8 // is not accessible by any other part of the program
9
10 f(); //< When enter the function call, gets initialized
```

Local Variables

- Unless declared `static`, each invocation has its own copy.

```
1 void f() {
2     float var1 = 5.5F;
3     float var2 = 1.5F;
4     // do something with var1, var2
5 }
6
7 f(); //< First call, var1, var2 are created
8 f(); //< Second call, NEW var1, var2 are created
```

Local Variables

- `static` variable, a single, statically allocated object represent that variable in **all** calls.

```
1 void f() {
2     // same variable for all function calls
3     static int counter = 0;
4
5     // Increment counter on each function call
6     counter++;
7 }
8
9 // at this point, f::counter has been statically
10 // allocated and accessible by any function call to f()
11
12 f(); //< Acess counter, counter == 1
13 f(); //< Acess same counter, counter ==2
```

Local Variables

```
1 #include <iostream>
2 using namespace std;
3 void Counter() {
4     static int counter = 0;
5     cout << "counter state = " << ++counter << endl;
6 }
7 int main() {
8     for (size_t i = 0; i < 5; i++) {
9         Counter();
10    }
11    return 0;
12 }
```

NACHO-STYLE Avoid if possible, read more at:
<https://isocpp.org/wiki/faq/ctors#static-init-order>

Local Variables

- Any local variable will be destroyed when the execution exit the **scope** of the function.

```
1 void f() {  
2     // Gets initialized when the execution reaches  
3     // the definition of f(), thus, this implementation  
4     int local_variable = 50;  
5 }
```

local_variable has been destroyed at this point, RIP.

Argument List

- **How** the function interact with external world
- They all have a **type**, and a **name** as well.
- They are also called **parameters**.
- Unless is declared as reference, a **copy** of the actual argument is passed to the function.

Argument List

```
1 void f(type arg1, type arg2) {
2     // f holds a copy of arg1 and arg2
3 }
4
5 void f(type& arg1, type& arg2) {
6     // f holds a referencnce of arg1 and arg2
7     // f could possibly change the content
8     // of arg1 or arg2
9 }
10
11 void f(const type& arg1, const type& arg2) {
12     // f can't change the content of arg1 nor arg2
13 }
14
15 void f(type arg1, type& arg2, const type& arg3);
```

Default arguments

- Functions can accept default arguments
- Only **set in declaration** not in definition
- **Pro**: simplify function calls
- **Cons**:
 - Evaluated upon every call
 - Values are hidden in declaration
 - Can lead to unexpected behavior when overused
- **GOOGLE-STYLE** Only use them when readability gets much better
- **NACHO-STYLE** Never use them

Example: default arguments

```
1 #include <iostream>
2 using namespace std;
3
4 string SayHello(const string& to_whom = "world") {
5     return "Hello " + to_whom + "!";
6 }
7
8 int main() {
9     // Will call SayHello using the default argument
10    cout << SayHello() << endl;
11
12    // This will override the default argument
13    cout << SayHello("students") << endl;
14    return 0;
15 }
```

Passing big objects

- By default in C++, objects are copied when passed into functions
- If objects are big it might be slow
- **Pass by reference** to avoid copy

```
1 void DoSmtH(std::string huge_string); // Slow.  
2 void DoSmtH(std::string& huge_string); // Faster.
```



Is the string still the same?

```
1 string hello = "some_important_long_string";  
2 DoSmtH(hello);
```

Unknown without looking into `DoSmtH()`!

Solution: use const references

- Pass **const** reference to the function
- Great speed as we pass a reference
- Passed object stays intact

```
1 void DoSmtth(const std::string& huge_string);
```

- Use **snake_case** for all function arguments
- Non-const refs are mostly used in older code written before C++ 11
- They can be useful but destroy readability
- **GOOGLE-STYLE** Avoid using non-const refs

Cost of passing by value

```
1 void pass_by_value(std::string huge_string) {
2     (void) huge_string;
3 }
4
5 // Pat attention to the -> "&" <- symbol
6 void pass_by_ref(std::string& huge_string) {
7     (void) huge_string;
8 }
9
10 static void PassByValue(benchmark::State& state) {
11     // Code inside this loop is measured repeatedly
12     std::string created_string("hello");
13     for (auto _ : state) {
14         pass_by_value(created_string);
15     }
16 }
17 BENCHMARK(PassByValue);
18
19 static void PassByRef(benchmark::State& state) {
20     // Code inside this loop is measured repeatedly
21     std::string created_string("hello");
22     for (auto _ : state) {
23         pass_by_ref(created_string);
24     }
25 }
26 BENCHMARK(PassByRef);
```

This function receive a copy of the value of the input string. Is the input string is big, this operation might cost a lot of time

This function receive a reference to the input_string. We will access the memory location where the input string is located

This function call will be evaluated in the benchmark

Pay attention to the benchmark names

<http://quick-bench.com/LqwBICOM3KrQE4tqupBtzqJmCdw>

Cost of passing by value

compiler = gcc-7.3 ▾

std = c++11 ▾

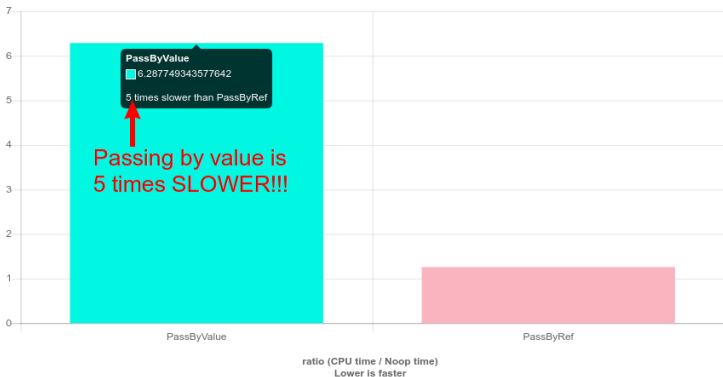
optim = None ▾

STL = libstdc++(GNU) ▾

Run Benchmark

Record disassembly

Clear cached results



Show Noop bar

`inline`

- `function` calls are expensive...
- Well, not **THAT** expensive though.
- If the function is rather small, you could help the compiler.
- `inline` is a **hint** to the compiler
 - should attempt to generate code for a call
 - rather than a function call.
- Sometimes the compiler do it anyways.

inline

```
1 inline int fac(int n) {  
2     if (n < 2) {  
3         return 2;  
4     }  
5     return n * fac(n - 1);  
6 }  
7  
8 int main() { return fac(10); }
```

Cehck it out:

<https://godbolt.org/z/amkfH4>

```
1 inline int fac(int n) {
2     if (n < 2) {
3         return 2;
4     }
5     return n * fac(n - 1);
6 }
7
8 int main() {
9     int fac0 = fac(0);
10    int fac1 = fac(1);
11    int fac2 = fac(2);
12    int fac3 = fac(3);
13    int fac4 = fac(4);
14    int fac5 = fac(5);
15    return fac0 + fac1 + fac2 + fac3 + fac4 + fac5;
16 }
```

Cehck it out:

<https://godbolt.org/z/EGd6aG>

C-style overloading

cosine

```
1 #include <math.h>
2
3 double cos(double x);
4 float cosf(float x);
5 long double cosl(long double x);
```

arctan

```
1 #include <math.h>
2
3 double atan(double x);
4 float atanf(float x);
5 long double atanl(long double x);
```

C-style overloading

usage

```
1 #include <math.h>
2 #include <stdio.h>
3
4 int main() {
5     double x_double = 0.0;
6     float x_float = 0.0;
7     long double x_long_double = 0.0;
8
9     printf("cos(0) = %f\n", cos(x_double));
10    printf("cos(0) = %f\n", cosf(x_float));
11    printf("cos(0) = %Lf\n", cosl(x_long_double));
12
13    return 0;
14 }
```

C++ style overloading

cosine

```
1 #include <cmath>
2
3 // ONE cos function to rule them all
4 double cos(double x);
5 float cos(float x);
6 long double cos(long double x);
```

arctan

```
1 #include <cmath>
2
3 double atan(double x);
4 float atan(float x);
5 long double atan(long double x);
```

C++ style overloading

usage

```
1 #include <cmath>
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     double x_double = 0.0;
7     float x_float = 0.0;
8     long double x_long_double = 0.0;
9
10    cout << "cos(0)=" << std::cos(x_double) << '\n';
11    cout << "cos(0)=" << std::cos(x_float) << '\n';
12    cout << "cos(0)=" << std::cos(x_long_double) << '\n';
13
14    return 0;
15 }
```

Function overloading

- Compiler infers a function from arguments
- Cannot overload based on return type
- Return type plays no role at all
- **GOOGLE-STYLE** Avoid non-obvious overloads

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4 string TypeOf(int) { return "int"; }
5 string TypeOf(const string&) { return "string";}
6 int main() {
7     cout << TypeOf(1) << endl;
8     cout << TypeOf("hello") << endl;
9     return 0;
10 }
```

Good Practices

- Break up complicated computations into meaningful chunks and name them.
- Keep the length of functions **small** enough.
- Avoid **unnecessary** comments.
- One function should achieve **ONE** task.
- If you can't pick a short name, then **split** functionality.
- Avoid **macros**.
 - If you must use it, use ugly names with lots of capital letters.

Good function example

```
1 #include <vector>
2 using namespace std;
3
4 vector<int> CreateVectorOfZeros(int size) {
5     vector<int> null_vector(size);
6     for (int i = 0; i < size; ++i) {
7         null_vector[i] = 0;
8     }
9     return null_vector;
10 }
11
12 int main() {
13     vector<int> zeros = CreateVectorOfZeros(10);
14     return 0;
15 }
```

Bad function example #1

```
1 #include <vector>
2 using namespace std;
3 vector<int> Func(int a, bool b) {
4     if (b) { return vector<int>(10, a); }
5     vector<int> vec(a);
6     for (int i = 0; i < a; ++i) { vec[i] = a * i; }
7     if (vec.size() > a * 2) { vec[a] /= 2.0f; }
8     return vec;
9 }
```



- Name of the function means nothing
- Names of variables mean nothing
- Function does not have a single purpose

Bad function example #2

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 vector<int> CreateVectorAndPrintContent(int size) {
6     vector<int> vec(size);
7     for (size_t i = 0; i < size; i++) {
8         vec[i] = 0;
9         cout << vec[i] << endl;
10    }
11    return vec;
12 }
13
14 int main() {
15     vector<int> zeros = CreateVectorAndPrintContent(5);
16     return 0;
17 }
```

Bad function example #2 fix

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 vector<int> CreateVector(int size) {
6     vector<int> vec(size);
7     for (size_t i = 0; i < size; i++) {
8         vec[i] = 0;
9     }
10    return vec;
11 }
12
13 void PrintVector(std::vector<int> vec) {
14     for (auto element : vec) {
15         cout << element << endl;
16     }
17 }
```

Bad function example #3

```
1 // function user will onle see the declaration
2 // and NOT the definition.
3 // It's imposible to know at this point any
4 // additional information about this function.
5 int SquareNumber(int num);
```

Bad function example #3



Bad function example #3

```
1 // function user will only see the declaration
2 // and NOT the definition.
3 // It's impossible to know at this point any
4 // additional information about this function.
5 int SquareNumber(int num);
```

```
1 int SquareNumber(int num) {
2     // by the way, you need to call your aunt
3     // at this point. otherwise the program will
4     // fail with error code 314.
5     CallYourAunt();
6
7     return num * num;
8 }
```

Namespaces

module1

```
namespace module_1 {  
    int SomeFunc() {}  
}
```

module2

```
namespace module_2 {  
    int SomeFunc() {}  
}
```

- Helps avoiding name conflicts
- Group the project into logical modules

Namespaces example

```
1 #include <iostream>
2
3 namespace fun {
4 int GetMeaningOfLife(void) { return 42; }
5 } // namespace fun
6
7 namespace boring {
8 int GetMeaningOfLife(void) { return 0; }
9 } // namespace boring
10
11 int main() {
12     std::cout << boring::GetMeaningOfLife() << std::endl
13               << fun::GetMeaningOfLife() << std::endl;
14     return 0;
15 }
```

Namespaces example 2

- We don't like `std::vector` at all
- Let's define our own vector Class

```
1 // @file: my_vector.hpp
2 namespace my_vec {
3 template <typename T>
4 class vector {
5     // ...
6 };
7 } // namespace my_vec
```

Namespaces example 2

```
1 #include <vector>
2 #include "my_vecor.hpp"
3 int main() {
4     std::vector<int> v1; // Standard vector.
5     vec::vector<int> v2; // User defined vector.
6
7     {
8         using std::vector;
9         vector<int> v3; // Same as std::vector
10        v1 = v3; // OK
11    }
12
13    {
14        using vec::vector;
15        vector<int> v4; // Same as vec::vector
16        v2 = v4; // OK
17    }
18 }
```

Avoid using namespace <name>

```
1 #include <cmath>
2 #include <iostream>
3 using namespace std; // std namespace is used
4
5 // Self-defined function power shadows std::pow
6 double pow(double x, int exp) {
7     double res = 1.0;
8     for (int i = 0; i < exp; i++) {
9         res *= x;
10    }
11    return res;
12 }
13
14 int main() {
15     cout << "2.0^2 = " << pow(2.0, 2) << endl;
16     return 0;
17 }
```

Namespace error

Error output:

```
1 /home/ivizzo/.../namespaces_error.cpp:13:26:  
2 error: call of overloaded 'pow(double&, int&)' is  
   ambiguous  
3 double res = pow(x, exp);  
4                 ^  
5 ...
```

Only use what you need

```
1 #include <cmath>
2 #include <iostream>
3 using std::cout; // Explicitly use cout.
4 using std::endl; // Explicitly use endl.
5
6 // Self-defined function power shadows std::pow
7 double pow(double x, int exp) {
8     double res = 1.0;
9     for (int i = 0; i < exp; i++) {
10         res *= x;
11     }
12     return res;
13 }
14
15 int main() {
16     cout << "2.0^2 = " << pow(2.0, 2) << endl;
17     return 0;
18 }
```

Namespaces Wrap Up

Use namespaces to avoid name conflicts

```
1 namespace some_name {  
2 <your_code>  
3 } // namespace some_name
```

Use using correctly

■ [good]

- `using my_namespace::myFunc;`
- `my_namespace::myFunc(...);`
- **Never** use `using namespace name` in `*.hpp` files
- Prefer using explicit `using` even in `*.cpp` files

Nameless namespaces

- **GOOGLE-STYLE** for namespaces:

<https://google.github.io/styleguide/cppguide.html#Namespaces>

- **GOOGLE-STYLE** If you find yourself relying on some constants in a file and these constants should not be seen in any other file, put them into a **nameless namespace** on the top of this file

```
1 namespace {  
2   const int kLocalImportantInt = 13;  
3   const float kLocalImportantFloat = 13.0f;  
4 } // namespace
```

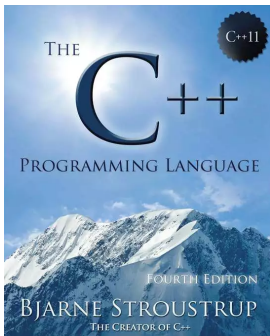
https://google.github.io/styleguide/cppguide.html#Unnamed_Namespaces_and_Static_Variables

Suggested Video



<https://youtu.be/cVC8bcV8zsQ>

References



- **Website:**

<http://www.stroustrup.com/4th.html>

References

- **Functions**

Stroustrup's book, chapter 12

- **Namesapces**

Stroustrup's book, chapter 14

- **cppreference**

<https://en.cppreference.com/w/cpp/language/function>

- **c-style strings**

<https://www.learncpp.com/cpp-tutorial/66-c-style-strings/>