Modern C++ for Computer Vision
Lecture 3: C++ Functions

Ignacio Vizzo, Rodrigo Marcuzzi, Cyrill Stachniss

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.*

Definition taken from: https://www.webopedia.com/TERM/F/function.html

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.*

Extract from: Section 12 of "The C++ Programming Language Book by Bjarne Stroustrup"

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**

Google style: https://google.github.io/styleguide/cppguide.html#Write_Short_Functions

Function Anatomy

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

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

Function Body

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

Function 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; }       // OK
5 void f4() { return 1; }      // error
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 }
```

Not always necessary:

```
1 int addition(int x, int y) {
2   return x + y;
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:

```
#include <iostream>
#include <tuple>

auto Foo() {
  return std::make_tuple("Super Variable", 5);
}

int main() {
  auto [name, value] = Foo();
  std::cout<<name <<" has value :"<<value<< std::endl;
  return 0;
}
```

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 accesisble anymore
9
10 int main() {
11   int out = MultiplyBy10(10);
12   cout << "out is " << out << endl;
13   return 0;
14 }
```

Return Type

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

int& MultiplyBy10(int num) {  // retval is created
  int retval = 0;
  retval = 10 * num;
  cout << "retval is " << retval << endl;
  return retval;
}  // retval is destroyed, it's not accesisble anymore

int main() {
  int out = MultiplyBy10(10);
  cout << "out    is " << out << endl;
  return 0;
}
```

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 defintion of f(), thus, this implementation
4    int local_variable = 50;
5  }
6
7  // at this point local_variable has not been intialized
8  // is not accessibly by any other part of the program
9
10 f();  //< When enter the function call, gets initalized
```

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 acessible 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 defintion 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 referecnce 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

Google style: `https://google.github.io/styleguide/cppguide.html#Default_Arguments`

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 DoSmth(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

Google style: https://google.github.io/styleguide/cppguide.html#Reference_Arguments

Cost of passing by value

```cpp
void pass_by_value(std::string huge_string) {
  (void) huge_string;
}

// Pat attention to the -> "&" <- symbol
void pass_by_ref(std::string& huge_string) {
  (void) huge_string;
}

static void PassByValue(benchmark::State& state) {
  // Code inside this loop is measured repeatedly
  std::string created_string("hello");
  for (auto _ : state) {
    pass_by_value(created_string);
  }
}
BENCHMARK(PassByValue);

static void PassByRef(benchmark::State& state) {
  // Code inside this loop is measured repeatedly
  std::string created_string("hello");
  for (auto _ : state) {
    pass_by_ref(created_string);
  }
}
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

`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); }
```

Check 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

Naive overloading

cosine

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

arctan

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

Naive overloading

usage

```
#include <cmath>
#include <iostream>

int main() {
  double x_double = 0.0;
  float x_float = 0.0;
  long double x_long_double = 0.0;

  cout << "cos(0)=" << cos(x_double) << '\n';
  cout << "cos(0)=" << cos(x_float) << '\n';
  cout << "cos(0)=" << cosl(x_long_double) << '\n';

  return 0;
}
```

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 }
```

Google style: https://google.github.io/styleguide/cppguide.html#Function_Overloading

Good Practices

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

Good function example

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

vector<int> CreateVectorOfZeros(int size);

int main() {
  vector<int> zeros = CreateVectorOfZeros(10);
  return 0;
}
```

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

Bad function example #3

Bad function example #3

```
// The user will only see the declaration
// and NOT the definition of the function.
// It's imposible to know any additional
// information about this function.
int SquareNumber(int num);
```

```
// Implementation
int SquareNumber(int num) {
  // this program will also play some music
  PlayMusic();

  return num * num;
}
```

Namespaces

module1

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

module2

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

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

https://en.cppreference.com/w/cpp/language/namespace

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::pow` at all
- Let’s define our own pow function

```
1 // @file: my_pow.hpp
2 namespace my_pow {
3   pow (...){
4     // ...
5   };
6 }  // namespace my_pow
```

Namespaces example 2

```
1  #include <cmath>
2  #include "my_pow.hpp"
3  int main() {
4    std_result = std::pow(2,3);    // Standard pow.
5    my_result = my_pow::pow(2,3); // User defined pow.
6
7    {
8      using std::pow;
9      result = pow(2,3);           // Same as std::pow
10   }
11
12   {
13     using my_pow::pow;
14     result = pow(2,3);           // Same as my_pow::pow
15   }
16 }
```

Avoid using namespace <name>

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

Namespace error

Error output:

```
1 namespaces_error.cpp: In function ‘int main()’:
2 namespaces_error.cpp:15:35: error: call of overloaded
    ‘pow(float&, float&)’ is ambiguous
3    15 |    cout << "2.0^2 = " << pow(x, exp) << endl;
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:

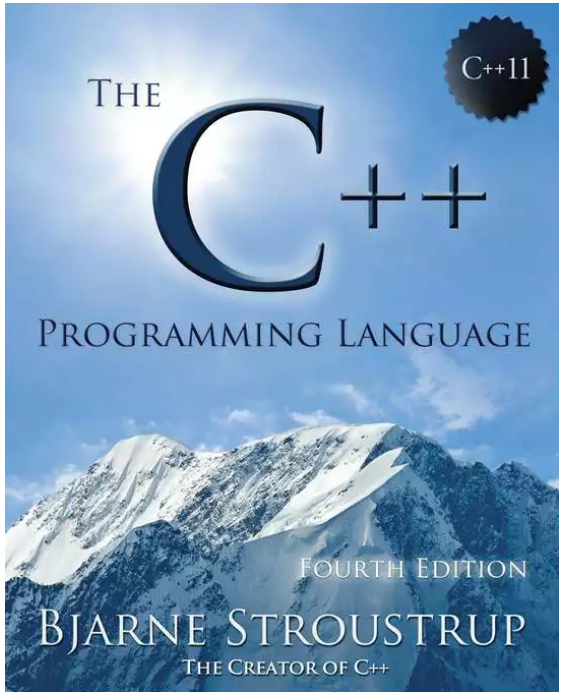
  https://google.github.io/styleguide/cppguide.html#Namespaces

- 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

References



- **Website:**

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

References

- **Functions**
  Stroustrp's book, chapter 12
- **Namesapces**
  Stroustrp's book, chapter 14
- **cppreference**

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