Modern C++ for Computer Vision and Image Processing

Igor Bogoslavskyi and Cyrill Stachniss

Outline

Intro to C++

- Variables and basic types
  - Built-in types
  - Strings
  - Vector and array
- Control structures
  - If statement
  - Switch statement
  - Loops

Git and homework submission

Declaring variables

Variable declaration always follows pattern:

```
<TYPE> <NAME> [ = <VALUE>];
```

- Every variable has a type
- Variables cannot change their type
- **Always initialize** variables if you can

```
1 int sad_uninitialized_var;
2 bool initializing_is_good = true;
```

Naming variables

- Name **must** start with a letter
- Give variables **meaningful names**
- Don’t be afraid to **use longer names**
- **Don’t include type** in the name
- **Don’t use negation** in the name
- `GOOGLE-STYLE` name variables in `snake_case`
  all lowercase, underscores separate words
- C++ is case sensitive:
  `some_var` is different from `some_Var`

Google naming rules: `https://google.github.io/styleguide/cppguide.html#General_Naming_Rules`

Built-in types

"Out of the box" types in C++:

```
1  bool this_is_fun = false;    // Boolean: true or false.
2  char carret_return = '\n';   // Single character.
3  int meaning_of_life = 42;    // Integer number.
4  short smaller_int = 42;      // Short number.
5  long bigger_int = 42;        // Long number.
6  float fraction = 0.01f;      // Single precision float.
7  double precise_num = 0.01;   // Double precision float.
8  auto some_int = 13;          // Automatic type [int].
9  auto some_float = 13.0f;     // Automatic type [float].
10 auto some_double = 13.0;     // Automatic type [double].
```

[Advanced] If curious read detailed info here:
http://en.cppreference.com/w/cpp/language/types

Operations on arithmetic types

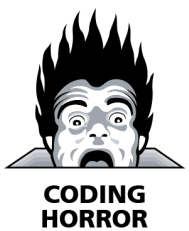
- All **character**, **integer** and **floating point** types are arithmetic
- Arithmetic operations: `+`, `-`, `*`, `/`
- Comparisons `<`, `>`, `<=`, `>=`, `==` return `bool`
- `a += 1` ⇔ `a = a + 1`, same for `-=`, `*=`, `/=`, etc.
- Avoid `==` for floating point types

Some additional operations

- Boolean variables have logical operations **or**: `||`, **and**: `&&`, **not**: `!`

```
1 bool is_happy = (!is_hungry && is_warm) || is_rich
```

- Additional operations on integer variables:
  - `/` is integer division: i.e. `7 / 3 == 2`
  - `%` is modulo division: i.e. `7 / 3 == 1`
  - **Increment** operator: `a++` ⇔ `++a` ⇔ `a += 1`
  - **Decrement** operator: `a--` ⇔ `--a` ⇔ `a -= 1`
  - Do not use de- increment operators within another expression, i.e. `a = (a++) + ++b`



Coding Horror image from Code Complete 2 book by Steve McConnell

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  int main() {
4    std::string hello = "Hello";
5    std::cout << "Type your name:" << std::endl;
6    std::string name = "";  // Init empty.
7    std::cin >> name;       // Read name.
8    std::cout << hello + ", " + name + "!" << std::endl;
9    return 0;
10 }
```

Use std::array for fixed size collections of items

- `#include <array>` to use `std::array`
- Store a **collection of items** of **same type**
- Create from data:
  `array<float, 3> arr = {1.0f, 2.0f, 3.0f};`
- Access items with `arr[i]`
  indexing starts with `0`
- Number of stored items: `arr.size()`
- Useful access aliases:
  - First item: `arr.front() == arr[0]`
  - Last item: `arr.back() == arr[arr.size() - 1]`

Use std::vector when number of items is unknown before-wise

- `#include <vector>` to use `std::vector`
- Vector is implemented as a **dynamic table**
- Access stored items just like in `std::array`
- Remove all elements: `vec.clear()`
- Add a new item in one of two ways:
  - `vec.emplace_back(value)` [preferred, c++11]
  - `vec.push_back(value)` [historically better known]
- **Use it! It is fast and flexible!**
  Consider it to be a default container to store collections of items of any same type

Optimize vector resizing

- Many `push_back`/`emplace_back` operations force vector to change its size many times
- `reserve(n)` ensures that the vector has enough memory to store `n` items
- The parameter `n` can even be approximate
- This is a very **important optimization**

```
1 std::vector<std::string> vec;
2 const int kIterNum = 100;
3 // Always call reserve when you know the size.
4 vec.reserve(kIterNum);
5 for (int i = 0; i < kIterNum; ++i) {
6   vec.emplace_back("hello");
7 }
```

Example vector

```
#include <string>
#include <vector>
#include <iostream>
using namespace std;
int main() {
  vector<int> numbers = {1, 2, 3};
  vector<string> names = {"Igor", "Cyrill"};
  names.push_back("another_string");
  cout << "First name: " << names.front() << endl;
  cout << "Last number: " << numbers.back() << endl;
  return 0;
}
```

Variables live in scopes

- There is a single global scope
- Local scopes start with { and ends with }
- All variables **belong to the scope** where they have been declared
- All variables die in the end of **their** scope
- This is the core of C++ memory system

```
1 int main() {  // Start of main scope.
2   float some_float = 13.13f;  // Create variable.
3   {  // New inner scope.
4     auto another_float = some_float;  // Copy variable.
5   }  // another_float dies.
6   return 0;
7 }  // some_float dies.
```

Any variable can be const

- Use `const` to declare a **constant**
- The compiler will guard it from any changes
- Keyword `const` can be used with **any** type
- `GOOGLE-STYLE` name constants in `CamelCase` starting with a small letter `k`:
  - `const float kImportantFloat = 20.0f;`
  - `const int kSomeInt = 20;`
  - `const std::string kHello = "hello";`
- `const` is part of type:
  variable `kSomeInt` has type `const int`
- **Tip:** declare everything `const` unless it **must** be changed

References to variables

- We can create a **reference** to any variable
- Use `&` to state that a variable is a reference
  - `float& ref = original_variable;`
  - `std::string& hello_ref = hello;`
- Reference is part of type:
  variable `ref` has type `float&`
- Whatever happens to a reference happens to the variable and vice versa
- Yields performance gain as references **avoid copying data**

Const with references

- References are fast but reduce control
- To avoid unwanted changes use `const`
  - `const float& ref = original_variable;`
  - `const std::string& hello_ref = hello;`

```
1  #include <iostream>
2  using namespace std;
3  int main() {
4    int num = 42;  // Name has to fit on slides
5    int& ref = num;
6    const int& kRef = num;
7    ref = 0;
8    cout << ref << " " << num << " " << kRef << endl;
9    num = 42;
10   cout << ref << " " << num << " " << kRef << endl;
11   return 0;
12 }
```

If statement

```
1 if (STATEMENT) {
2   // This is executed if STATEMENT == true
3 } else if (OTHER_STATEMENT) {
4   // This is executed if:
5   // (STATEMENT == false) && (OTHER_STATEMENT == true)
6 } else {
7   // This is executed if neither is true
8 }
```

- Used to conditionally execute code
- All the `else` cases can be omitted if needed
- `STATEMENT` can be **any boolean expression**

Switch statement

```
switch(STATEMENT) {
  case CONST_1:
    // This runs if STATEMENT == CONST_1.
    break;
  case CONST_2:
    // This runs if STATEMENT == CONST_2.
    break;
  default:
    // This runs if no other options worked.
}
```

- Used to conditionally execute code
- Can have many `case` statements
- `break` exits the `switch` block
- `STATEMENT` usually returns `int` or `enum` value

While loop

```
while (STATEMENT) {
  // Loop while STATEMENT == true.
}
```

Example `while` loop:

```
bool condition = true;
while (condition) {
  condition = /* Magically update condition. */
}
```

- Usually used when the exact number of iterations is unknown before-wise
- Easy to form an endless loop by mistake

For loop

```
for (INITIAL_CONDITION; END_CONDITION; INCREMENT) {
  // This happens until END_CONDITION == false
}
```

Example `for` loop:

```
for (int i = 0; i < COUNT; ++i) {
  // This happens COUNT times.
}
```

- In C++ `for` loops are *very* fast. Use them!
- Less flexible than `while` but less error-prone
- Use `for` when number of iterations is fixed and `while` otherwise

Range for loop

- Iterating over a standard containers like `array` or `vector` has simpler syntax
- Avoid mistakes with indices
- Show intent with the syntax
- Has been added in C++11

```
1 for (const auto& value : container) {
2   // This happens for each value in the container.
3 }
```

Exit loops and iterations

- We have control over loop iterations
- Use `break` to exit the loop
- Use `continue` to skip to next iteration

```
1 while (true) {
2   int i = /* Magically get new int. */
3   if (i % 2 == 0) {
4     cerr << i << endl;
5   } else {
6     break;
7   }
8 }
```

- Free software for distributed version control
- **synchronizes** local and remote files
- **stores a history** of all changes

What is synchronized?

- **Local** files on a computer
- **Remote** Files in the repository
- We are using a **Gitlab** server

Example repository:

https://gitlab.igg.uni-bonn.de/teaching/cpp-homeworks-2018

Typical workflow

Cloning a repository:

- `git clone <repo_url> <local_folder>`

In `<local_folder>`:

- Change files
- `git add <files>`
- `git commit -am 'descriptive message'`
- `git push origin master`

Git — the simple guide:

http://rogerdudler.github.io/git-guide/

Submit homeworks through Git

- Log in to https://gitlab.igg.uni-bonn.de/
- Request access to `cpp-2018` group:
  https://gitlab.igg.uni-bonn.de/students/cpp-2018
- Fork the base homework repository:
  https://gitlab.igg.uni-bonn.de/Teaching/cpp-homeworks-2018
- To **fork** a repository in Git means to create a copy of the repository for your user

Submit homeworks through Git

The address of your fork will be:

`/<your_name>/cpp-homeworks-2018`

instead of:

`/teaching/cpp-homeworks-2018`

To enable homework checks, from your fork:

- Settings > Members > Select members to invite
- Pick `@hw_bot`  with `developer` rights
- This bot updates the Wiki in your project with evaluation of your homework
- Now push anything into the repo:

```
git push origin master
```

How to see evaluation results

- Your repository has a **Wiki** page
- In a couple of minutes after a `push` open the wiki page
- Example look:

Test results

Homework Name	Task Name	Test Name	Result
Bash and C++ intro	Guessing game	Build Succeeded	✓
	Simple Bash	Test 1	✓

With 💙 from homework bot 🤖

References

- **Cpp Core Guidelines:**
  https://github.com/isocpp/CppCoreGuidelines
- **Google Code Styleguide:**
  https://google.github.io/styleguide/cppguide.html
- **Git guide:**
  http://rogerdudler.github.io/git-guide/
- **C++ Tutorial:**
  http://www.cplusplus.com/doc/tutorial/
- Book: **Code Complete 2** by Steve McConnell