Modern C++ for Computer Vision and Image Processing

Igor Bogoslavskyi

Outline

Google Tests

Namespaces

Classes

Use GTest to test your functions

- Catch bugs early to fix them with less pain
- Testing is crucial to catch bugs early
- Tested functions are easier to trust
- For every function write at least **two tests**
  - One for normal cases
  - One for extreme cases
- **Make writing tests a habit**

How do tests look?

- A single dummy Google test:

```
TEST(TestModule, FunctionName) {
  EXPECT_EQ(4, FunctionName());
}
```

- Successful output:

```
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from TesModule
[ RUN      ] TesModule.FunctionName
[       OK ] TesModule.FunctionName (0 ms)
[----------] 1 test from TesModule (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran.
[  PASSED  ] 1 test.
```

Add GTests with CMake

- Install GTest source files (build them later):
  `sudo apt install libgtest-dev`
- Add folder `tests` to your CMake project:

```
# Must be in the top-most CMakeLists.txt file.
enable_testing()
# Outsource tests to another folder.
add_subdirectory(tests)
```

Configure tests

```cmake
# Add gtest sources folder. Provides gtest, gtest_main.
add_subdirectory(/usr/src/gtest
                 ${PROJECT_BINARY_DIR}/gtest)
include(CTest) # Include testing cmake package.
# Set binary name for convenience.
set(TEST_BINARY ${PROJECT_NAME}_test)
# This is an executable that runs the tests.
add_executable(${TEST_BINARY} test_tools.cpp)
# Link the executable to needed libraries.
target_link_libraries(${TEST_BINARY}
  tools              # Library we are testing
  gtest gtest_main   # GTest libraries
)
# Add gtest to be able to run ctest
add_test(
  NAME ${TEST_BINARY}
  COMMAND ${EXECUTABLE_OUTPUT_PATH}/${TEST_BINARY})
```

Run your tests

- Build your code just like before
- Add one **additional step** after building
  1. `cd <project_folder>`
  2. `mkdir build`
  3. `cd build`
  4. `cmake ..`
  5. `make`
  6. `ctest -VV`

Namespaces

module1

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

module2

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

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

Namespaces example

```cpp
#include <iostream>

namespace fun {
int GetMeaningOfLife() { return 42; }
}  // namespace fun

namespace boring {
int GetMeaningOfLife() { return 0; }
}  // namespace boring

int main() {
  std::cout << "The answer to everything is not "
            << boring::GetMeaningOfLife() << " but "
            << fun::GetMeaningOfLife() << std::endl;
  return 0;
}
```

Avoid `using namespace <name>`

```cpp
#include <cmath>
#include <iostream>
using namespace std;  // std namespace is used
// Self-defined function power shadows std::pow
double pow(double x, int exp) {
  double res = 1.0;
  for (int i = 0; i < exp; i++) { res *= x; }
  cout << "Our cool power function\n";
  return (res);
}
int main() {
  double x = 2.0;
  int power = 2;
  double res = pow(x, power);
  cout << x << " ^ " << power << " = " << res << endl;
  return 0;
}
```

Namespace error

Error output:

```
/home/igor/.../namespaces_error.cpp:13:26:
error: call of overloaded ‘pow(double&, int&)’ is ambiguous
double res = pow(x, exp);
                       ^
...
```

Only use what you need

```cpp
#include <cmath>
#include <iostream>
using std::cout;  // Explicitly use cout.
using std::endl;  // Explicitly use endl.
// Self-defined function power shadows std::pow
double pow(double x, int exp) {
  double res = 1.0;
  for (int i = 0; i < exp; i++) { res *= x; }
  cout << "Our cool power function\n";
  return (res);
}
int main() {
  double x = 2.0;
  int power = 2;
  double res = pow(x, power);
  cout << x << " ^ " << power << " = " << res << endl;
  return 0;
}
```

Namespaces Wrap Up

Use namespaces to avoid name conflicts

```
namespace some_name {
<your_code>
}  // namespace some_name
```

Use using correctly

- **[good]**
  - `using my_namespace::myFunc;`
  - `my_namespace::myFunc(…);`
- **Never** use `using namespace name` in `*.h` files
- Prefer using explicit `using` even in `*.cpp` files

Nameless namespaces

If you find yourself relying on some contstants 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

```
namespace {
const int kLocalImportantInt = 13;
const float kLocalImportantFloat = 13.0f;
}  // namespace
```

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

Create new types with classes and structs

- Classes are used to **encapsulate data** along with methods to process them
- Every `class` or `struct` defines a new type
- **Terminology**:
  - **Type** or **class** to talk about the defined type
  - A variable of such type is an **instance of class** or an **object**
- Classes allow C++ to be used as an **O**bject **O**riented **P**rogramming language
- `string`, `vector`, etc. are all classes

http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c-classes-and-class-hierarchies

Example class definition

```cpp
class Image {
 public:
  Image(const std::string& file_name);
  void Draw();
 private:
  int rows_ = 0;
  int cols_ = 0;
};
// Implementation omitted here.
int main() {
  Image image("some_image.pgm");
  image.Draw();
  return 0;
}
```

Classes syntax

- Definition starts with the keyword `class`
- Classes have **three access modifiers**: `private`, `protected` and `public`
- By default everything is `private`
- Classes can contain data and functions
- Access members with a `"."`
- Have two types of **special functions**:
  - **Contructors**: called upon **creation** of an instance of the class
  - **Destructor**: called upon **destruction** of an instance of the class
- GOOGLE-STYLE Use `CamelCase` for class name

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

What about structs?

- Definition starts with the keyword `struct`:

```cpp
struct ExampleStruct {
  Type value;
  Type value;
  Type value;
  // No functions!
};
```

- `struct` is a `class` where everything is `public`
- GOOGLE-STYLE Use `struct` as a **simple data container**, if it needs a function it should be a `class` instead

https://google.github.io/styleguide/cppguide.html#Structs_vs._Classes

Always initialize structs using braced initialization

```cpp
#include <iostream>
#include <string>
using namespace std;
// Define a structure.
struct NamedInt {
  int num;
  string name;
};
void PrintStruct(const NamedInt& s) {
  cout << s.name << " " << s.num << endl;
}
int main(int argc, char const* argv[]) {
  NamedInt var = {1, "hello"};
  PrintStruct(var);
  PrintStruct({10, "world"});
  return 0;
}
```

Data stored in a class

- Classes can store data of any type
- `GOOGLE-STYLE` All data must be `private`
- `GOOGLE-STYLE` Use `snake_case_` with a trailing `"_"` for `private` data members
- Data should be **set in the Constructor**
- **Cleanup data in the Destructor** if needed

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

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

Constructors and Destructor

- Classes always have **at least one Constructor** and **exactly one Destructor**
- Constructors crash course:
  - Are functions with no return type
  - Named exactly as the class
  - There can be many constructors
  - **If there is no explicit constructor an implicit default constructor will be generated**
- Destructor for class `SomeClass`:
  - Is a function named `~SomeClass()`
  - Last function called in the lifetime of an object
  - Generated automatically if not explicitly defined

Many ways to create instances

```cpp
class SomeClass {
 public:
  SomeClass();                // Default constructor.
  SomeClass(int a);           // Custom constructor.
  SomeClass(int a, float b);  // Custom constructor.
  ~SomeClass();               // Destructor.
};
// How to use them?
int main() {
  SomeClass var_1;               // Default constructor
  SomeClass var_2(10);           // Custom constructor
  // Type is checked when using {} braces. Use them!
  SomeClass var_3{10};           // Custom constructor
  SomeClass var_4 = {10};        // Same as var_3
  SomeClass var_5{10, 10.0};     // Custom constructor
  SomeClass var_6 = {10, 10.0};  // Same as var_5
  return 0;
}
```

Setting and getting data

- Use **initializer list** to initialize data
- Name getter functions as the private member they return
- Make getters `const`
- **Avoid setters**, set data in the constructor

```
class Student {
 public:
  Student(int id, string name): id_{id}, name_{name} {}
  int id() const { return id_; }
  const string& name() const { return name_; }
 private:
  int id_;
  string name_;
};
```

Const correctness

- `const` after function states that this function **does not change the object**
- Mark all functions that **should not** change the state of the object as `const`
- Ensures that we can pass objects by a `const` reference and still call their functions
- Substantially reduces number of errors

Typical const error

```cpp
#include <string>
#include <iostream>
using namespace std;
class Student {
 public:
  Student(string name): name_{name} {}
  const string& name() { return name_; }
 private:
  string name_;
};
void Print(const Student& student) {
  cout << "Student: " << student.name() << endl;
}
```

```
error: passing "const Student" as "this" argument discards qualifiers [-fpermissive]
   cout << "Student: " << student.name() << endl;
                                       ^
```

Declaration and definition

- Data members belong to declaration
- Class methods can be defined elsewhere
- Class name becomes part of function name

```
// Declare class.
class SomeClass {
 public:
  SomeClass();
  int var() const;
 private:
  void DoSmth();
  int var_ = 0;
};
// Define all methods.
SomeClass::SomeClass() {}
int SomeClass::var() const { return var_; }
void SomeClass::DoSmth() {}
```

Always initialize members for classes

- C++11 allows to initialize variables in-place
- Do not initialize them in the constructor
- No need for an explicit default constructor

```
class Student {
 public:
  // No need for default constructor.
  // Getters and functions omitted.
 private:
  int earned_points_ = 0;
  float happiness_ = 1.0f;
};
```

- **Note:** Leave the members of `structs` uninitialized as defining them forbids using brace initialization

Classes as modules

- Prefer encapsulating information that belongs together into a class
- **Separate declaration and definition** of the class into header and source files
- Typically, class `SomeClass` is declared in `some_class.h` and is defined in `some_class.cpp`

References

- **Const correctness:**
  https://isocpp.org/wiki/faq/const-correctness
- **Google Test primer:**
  https://goo.gl/JzFBYh `[shortened]`