Modern C++ for Computer Vision and Image Processing

Igor Bogoslavskyi

Outline

Classes

Polymorphism

I/O

Stringstreams

CMake find_package

Polymorphism

> From Greek **polys**, "many, much" and **morphē**, "form, shape"
>
> -Wiki

- Allows morphing derived classes into their base class type:

```
const Base& base = Derived(…)
```

When is it useful?

- Allows encapsulating the implementation inside a class only asking it to conform to a common interface
- Often used for:
  - Working with all children of some Base class in unified manner
  - Enforcing an interface in multiple classes to force them to implement some functionality
  - In **strategy** pattern, where some complex functionality is outsourced into separate classes and is passed to the object in a modular fashion

Creating a class hierarchy

- Sometimes classes must form a hierarchy
- Distinguish between **is a** and **has a** to test if the classes should be in one hierarchy:
  - Square **is a** Shape: can inherit from Shape
  - Student **is a** Human: can inherit from Human
  - Car **has a** Wheel: should **not** inherit each other
- Prefer shallow hierarchies
- `GOOGLE-STYLE` **Prefer composition**, i.e. including an object of another class as a member of your class

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

```cpp
#include <iostream>
using std::cout; using std::endl;
class Rect {
 public:
  Rect(int w, int h) : w_{w}, h_{h} {}
  virtual void Print() const {
    cout << "Rect: " << w_ << " x " << h_ << endl;
  }
 protected:
  int w_ = 0; int h_ = 0;
};
struct Square : public Rect {  // Should be a class.
  explicit Square(int size) : Rect{size, size} {}
  void Print() const override {
    cout << "Square: " << w_ << " x " << h_ << endl;
  }
};
void Print(const Rect& rect) { rect.Print(); }
int main() {
  Print(Square(10)); Print(Rect(10, 20));
  return 0;
}
```

Using interfaces

- Use interfaces when you must **enforce** other classes to implement some functionality
- Allow thinking about classes in terms of **abstract functionality**
- **Hide implementation** from the caller
- Allow to easily extend functionality by simply adding a new class

```cpp
#include <iostream>
using std::cout;
using std::endl;
struct Printable {  // Saving space. Should be a class.
  virtual void Print() const = 0;
};
struct A : public Printable {
  void Print() const override { cout << "A" << endl; }
};
struct B : public Printable {
  void Print() const override { cout << "B" << endl; }
};
void Print(const Printable& var) { var.Print(); }
int main() {
  Print(A());
  Print(B());
  return 0;
}
```

Using strategy pattern

- If a class relies on complex external functionality use strategy pattern
- Allows to **add/switch functionality** of the class without changing its implementation
- All strategies must conform to one strategy interface

```cpp
#include <iostream>
using std::cout; using std::endl;
struct Strategy { // Saving space, should be classes.
  virtual void Print() const = 0;
};
struct StrategyA : public Strategy {
  void Print() const override { cout << "A" << endl; }
};
struct StrategyB : public Strategy {
  void Print() const override { cout << "B" << endl; }
};
struct MyStruct {
  MyStruct(const Strategy& s): strategy_(s) {}
  void Print() const { strategy_.Print(); }
  const Strategy& strategy_;
};
int main() {
  // Create a local var of MyStruct and call its Print
  MyStruct(StrategyA()).Print();
  MyStruct(StrategyB()).Print();
}
```

Do not overuse it

- Only use these patterns when you need to
- If your class should have a single method for some functionality and will never need another implementation don’t make it virtual
- Used mostly to **avoid copying code** and to **make classes smaller** by moving some functionality out

Reading and writing to files

- Use streams from STL
- Syntax similar to `cerr, cout`

```
1 #include <fstream>
2 using std::string;
3 using Mode = std::ios_base::openmode;
4 // ifstream: stream for input from file
5 std::ifstream f_in(string& file_name, Mode mode);
6 // ofstream: stream for output to file
7 std::ofstream f_out(string& file_name, Mode mode);
8 // stream for input and output to file
9 std::fstream f_in_out(string& file_name, Mode mode);
```

There are many modes under which a file can be opened

Mode	Meaning
`ios_base::app`	append output
`ios_base::ate`	seek to EOF when opened
`ios_base::binary`	open the file in binary mode
`ios_base::in`	open the file for reading
`ios_base::out`	open the file for writing
`ios_base::trunc`	overwrite the existing file

Regular columns

Use it when:

- The file constains organized data
- Every line has to have all columns

```
1 1 2.34 One 0.21
2 2 2.004 two 0.23
3 3 -2.34 string 0.22
```

O.K.

```
1 1 2.34 One word 0.21
2 2 2.004 two 0.23
3 3 -2.34 string 0.22
```

Fail

```
1 1 2.34 One 0.21
2 2 2.004 two
3 3 -2.34 string 0.22
```

Reading from ifstream

```
1  #include <fstream>  // For the file streams.
2  #include <iostream>
3  #include <string>
4  using namespace std;  // Saving space.
5  int main() {
6    int i;
7    double a, b;
8    string s;
9    // Create an input file stream.
10   ifstream in("test_cols.txt", ios_base::in);
11   // Read data, until it is there.
12   while (in >> i >> a >> s >> b) {
13     cerr << i << ", " << a << ", "
14          << s << ", " << b << endl;
15   }
16   return (0);
17 }
```

Reading files one line at a time

- Bind every line to a `string`
- Afterwards parse the string

```
1 ================================
2 HEADER
3 a = 4.5
4 filename = /home/igor/.bashrc
5 ================================
6 2.34
7 1 2.23
8 ER SIE ES
```

```cpp
#include <fstream>  // For the file streams.
#include <iostream>
using namespace std;
int main() {
  string line, file_name;
  ifstream input("test_bel.txt", ios_base::in);
  // Read data line-wise.
  while (getline(input, line)) {
    cout << "Read: " << line << endl;
    // String has a find method.
    string::size_type loc = line.find("filename", 0);
    if (loc != string::npos) {
      file_name = line.substr(line.find("=", 0) + 1,
                              string::npos);
    }
  }
  cout << "Filename found: " << file_name << endl;
  return (0);
}
```

Writing into text files

With the same syntax as `cerr` und `cout` streams, with `ofstream` we can write directly into files

```
#include <iomanip>  // For setprecision.
#include <fstream>
using namespace std;
int main() {
  string filename = "out.txt";
  ofstream outfile(filename);
  if (!outfile.is_open()) { return EXIT_FAILURE; }
  double a = 1.123123123;
  outfile << "Just string" << endl;
  outfile << setprecision(20) << a << endl;
  return 0;
}
```

String streams

Already known streams:

- Standard output: `cerr`, `cout`
- Standard input: `cin`
- Filestreams: `fstream`, `ifstream`, `ofstream`

New type of stream: `stringstream`

- Combine `int`, `double`, `string`, etc. into a single `string`
- Break up `strings` into `int`, `double`, `string` etc.

```cpp
#include <iomanip>
#include <iostream>
#include <sstream>
using namespace std;
int main() {
  stringstream s_out;
  string ext = ".txt", file_name = "";
  for (int i = 0; i < 500; ++i) {
    // Combine variables into a stringstream.
    s_out << setw(5) << setfill('0') << i << ext;
    file_name = s_out.str();  // Get a string.
    s_out.str("");  // Empty stream for next iteration.
    cerr << file_name << endl;
  }
  stringstream s_in(file_name);
  int i; string rest;
  s_in >> i >> rest;
  cerr << "Number: " << i << " rest is: " << rest;
  return 0;
}
```

CMake `find_path` and `find_library`

- We can use an external library
- Need headers and binary library files
- There is an easy way to find them
- **Headers:**

```
find_path(SOME_PKG_INCLUDE_DIR include/some_file.h
          <path1> <path2> ...)
include_directories(${SOME_PKG_INCLUDE_DIR})
```

- **Libraries:**

```
find_library(SOME_LIB
             NAMES <some_lib>
             PATHS <path1> <path2> ...)
target_link_libraries(target ${SOME_LIB})
```

find_package

- find_package calls multiple find_path and find_library functions
- To use find_package(<pkg>) CMake must have a file Find<pkg>.cmake in CMAKE_MODULE_PATH folders
- Find<pkg>.cmake defines which libraries and headers belong to package <pkg>
- Pre-defined for most popular libraries, e.g. OpenCV, libpng, etc.

CMakeLists.txt

```
1  cmake_minimum_required(VERSION 2.8)
2  project(first_project)
3
4  # CMake will search here for Find<pkg>.cmake files
5  SET(CMAKE_MODULE_PATH
6      ${PROJECT_SOURCE_DIR}/cmake_modules)
7
8  # Search for Findsome_pkg.cmake file and load it
9  find_package(some_pkg)
10
11 # Add the include folders from some_pkg
12 include_directories(${some_pkg_INCLUDE_DIRS})
13
14 # Add the executable "main"
15 add_executable(main small_main.cpp)
16 # Tell the linker to bind these binary objects
17 target_link_libraries(main ${some_pkg_LIBRARIES})
```

cmake_modules/Findsome_pkg.cmake

```
1  # Find the headers that we will need
2  find_path(some_pkg_INCLUDE_DIRS include/some_lib.h
3            <FOLDER_WHERE_TO_SEARCH>)
4  message(STATUS "headers: ${some_pkg_INCLUDE_DIRS}")
5
6  # Find the corresponding libraries
7  find_library(some_pkg_LIBRARIES
8               NAMES some_lib_name
9               PATHS <FOLDER_WHERE_TO_SEARCH>)
10 message(STATUS "libs: ${some_pkg_LIBRARIES}")
```

References

- **Fluent C++: structs vs classes:**
  https://goo.gl/NFo8HP [shortened]