Modern C++ for Computer Vision and Image Processing

Igor Bogoslavskyi
Outline

Generic programming
  Template functions
  Template classes

Iterators

Error handling

Program input parameters

OpenCV
  cv::Mat
  cv::Mat I/O
  SIFT Extraction
  FLANN in OpenCV
  OpenCV with CMake
Generic programming: separate algorithms from the data type

Cup holds any type $T$, e.g. Coffee or Tea
Template functions

Generic programming uses keyword template

```c++
template <typename T, typename S>
T awesome_function(const T& var_t, const S& var_s) {
    // some dummy implementation
    T result = var_t;
    return result;
}
```

T and S can be any type that is:
- Copy constructable
- Assignable
- Is defined (for custom classes)
Explicit type

If the data type cannot be determined by the compiler, we must define it **ourselves**

```cpp
1 // Function definition.
2 template <typename T>
3 T DummyFuncion() {
4   T result;
5   return result;
6 }
7
8 int main(int argc, char const *argv[]) {
9   DummyFuncion<int>();
10  DummyFuncion<double>();
11  return 0;
12 }
```
Template classes

- Similar syntax to template functions
- Use template type anywhere in class

```
```template <class T>
```class MyClass {
  ```public:
  ```MyClass(const T& smth) : smth_(smth) {} 
  ```private:
  T smth_; 
```
```
```
int main(int argc, char const* argv[]) {
  ```MyClass<int> my_object(10);
  MyClass<double> my_double_object(10.0);
  ```return 0;
```
Template specialisation

- We can specialize for a type
- Works for functions and classes alike

```cpp
// Function definition.
template <typename T>
T DummyFuncion() {
    T result;
    return result;
}

template <>
int DummyFuncion() {
    return 42;
}

int main() {
    DummyFuncion<int>();
    DummyFuncion<double>();
    return 0;
}
```
Template meta programming

- Templates are used for Meta programming
- The compiler will generate concrete instances of generic classes based on the classes we want to use
- If we create `MyClass<int>` and `MyClass<float>` the compiler will generate two different classes with appropriate types instead of template parameter
Template classes headers/source

- Concrete template classes are generated instantiated at compile time
- Linker does not know about implementation
- There are three options for template classes:
  - Declare and define in header files
  - Declare in `NAME.h` file, implement in `NAME.hpp` file, add `#include <NAME.hpp>` in the end of `NAME.h`
  - Declare in `*.h` file, implement in `*.cpp` file, in the end of the `*.cpp` add explicit instantiation for types you expect to use
- Read more about it:
Iterators

STL uses iterators to access data in containers

- Iterators are similar to pointers
- Allow quick navigation through containers
- Most algorithms in STL use iterators
- Access current element with \(*iter\)
- Accepts \(\rightarrow\) alike to pointers
- Move to next element in container \(iter++\)
- Prefer range-based for loops
- Compare iterators with \(==, !=, <\)
- Pre-defined iterators: \(obj.begin()\), \(obj.end()\)
```cpp
#include <iostream>
#include <map>
#include <vector>
using namespace std;

int main() {
    // Vector iterator.
    vector<double> x = {{1, 2, 3}};
    for (auto it = x.begin(); it != x.end(); ++it) {
        cout << *it << endl;
    }

    // Map iterators
    map<int, string> m = {{1, "hello"}, {2, "world"}};
    map<int, string>::iterator m_it = m.find(1);
    cout << m_it->first <<":" << m_it->second << endl;
    if (m.find(3) == m.end()) {
        cout << "Key 3 was not found\n";
    }
    return 0;
}
```
Error handling with exceptions

- We can “throw” an exception if there is an error
- STL defines classes that represent exceptions. Base class: exception
- To use exceptions: #include <stdexcept>
- An exception can be “caught” at any point of the program (try – catch) and even “thrown” further (throw)
- The constructor of an exception receives a string error message as a parameter
- This string can be called through a member function what()
throw exceptions

**Runtime Error:**

```cpp
// if there is an error
if (badEvent) {
    string msg = "specific error string";
    // throw error
    throw runtime_error(msg);
}

... some cool code if all ok ...
```

**Logic Error:** an error in logic of the user

```cpp
throw logic_error(msg);
```
**catch exceptions**

- If we expect an exception, we can “catch” it
- Use **try - catch** to catch exceptions

```cpp
try {
    // some code that can throw exceptions z.B.
    x = someUnsafeFunction(a, b, c);
}
// we can catch multiple types of exceptions
catch ( runtime_error &ex ) {
    cerr << "Runtime error: " << ex.what() << endl;
} catch ( logic_error &ex ) {
    cerr << "Logic error: " << ex.what() << endl;
} catch ( exception &ex ) {
    cerr << "Some exception: " << ex.what() << endl;
} catch ( ... ) { // all others
    cerr << "Error: unknown exception" << endl;
}
```
Intuition

- Only used for "exceptional behavior"
- **Often misused**: e.g. wrong parameter should not lead to an exception
- **GOOGLE-STYLE** Don’t use exceptions
- http://www.cplusplus.com/reference/exception/

https://google.github.io/styleguide/cppguide.html#Exceptions
Program input parameters

- Originate from the declaration of main function
- Allow passing arguments to the binary
- `int main(int argc, char const *argv[]);`
- `argc` defines number of input parameters
- `argv` is an array of string parameters
- By default:
  - `argc == 1`
  - `argv == "<binary_path>"`

Program input parameters

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

int main(int argc, char const *argv[]) {
    cout << "Got " << argc << " params\n";
    string program_name = argv[0];
    cout << "Program: " << program_name << endl;
    for (int i = 1; i < argc; ++i) {
        cout << "Param: " << argv[i] << endl;
    }
    return 0;
}
```
Using for type aliasing

- Use word `using` to declare new types from existing and to create type aliases
- **Basic syntax:** `using NewType = OldType;`
- `using` is a versatile word
- When used outside of functions declares a new type alias
- When used in function creates an alias of a type available in the current scope
#include <array>
#include <memory>

template <class T, int SIZE>
struct Image {
    // Can be used in classes.
    using Ptr = std::unique_ptr<Image<T, SIZE>>;
    std::array<T, SIZE> data;
};

// Can be combined with "template".
template <int SIZE>
using Imagef = Image<float, SIZE>;

int main() {
    // Can be used in a function for type aliasing.
    using Image3f = Imagef<3>;
    auto image_ptr = Image3f::Ptr(new Image3f);
    return 0;
}
OpenCV

- Popular library for **Image Processing**
- We will be using **version 2** of OpenCV
- We will be using just a small part of it
- `#include <opencv2/opencv.hpp>` to use all functionality available in OpenCV
- Namespace `cv::`
Data types

- OpenCV uses **own types**
- OpenCV trusts you to pick the correct type
- Names of types follow pattern `CV_<bit_count><identifier><num_of_channels>`
  - **Example**: RGB image is `CV_8UC3`: 8-bit unsigned char with 3 channels for RGB
  - **Example**: Grayscale image is `CV_8UC1`: single 8-bit unsigned char for intensity
- Better to use `DataType`
  - **Example**: `DataType<uint>::type == CV_8UC1`
Basic Matrix Type

- Every image is a `cv::Mat`, for “Matrix”
- Mat image(rows, cols, DataType, Value);
- Mat_<T> image(rows, cols, Value);
- Initialize with `zeros`:
  1. cv::Mat image = cv::Mat::zeros(10, 10, CV_8UC3);
  2. using Matf = cv::Mat_<float>;
  3. Matf image_float = Matf::zeros(10, 10);

- Get type identifier with `image.type()`;
- Get size with `image.rows, image.cols`
- I/O:
  - Read image with `imread`
  - Write image with `imwrite`
  - Show image with `imshow`
  - Detects I/O method from extension
cv::Mat is a shared pointer

It does not use std::shared_ptr but follows the same principle of reference counting

```cpp
#include <opencv2/opencv.hpp>
#include <iostream>

int main() {
    using Matf = cv::Mat_<float>;
    Matf image = Matf::zeros(10, 10);
    Matf image_no_copy = image;  // Does not copy!
    image_no_copy.at<float>(5, 5) = 42.42f;
    std::cout << image.at<float>(5, 5) << std::endl;
    Matf image_copy = image.clone();  // Copies image.
    image_copy.at<float>(1, 1) = 42.42f;
    std::cout << image.at<float>(1, 1) << std::endl;
}
```

```bash
c++ -std=c++11 -o copy copy.cpp \
`pkg-config --libs --cflags opencv`
```
**imread**

- Read image from file
- `Mat imread(const string& file, int mode=1)`
- **Different modes:**
  - unchanged: `CV_LOAD_IMAGE_UNCHANGED < 0`
  - 1 channel: `CV_LOAD_IMAGE_GRAYSCALE == 0`
  - 3 channels: `CV_LOAD_IMAGE_COLOR > 0`

```cpp
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace cv;

int main() {
    Mat i1 = imread("logo_opencv.png", CV_LOAD_IMAGE_GRAYSCALE);
    Mat_<uint8_t> i2 = imread("logo_opencv.png", CV_LOAD_IMAGE_GRAYSCALE);
    std::cout << (i1.type() == i2.type()) << std::endl;
    return 0;
}
```
### imwrite

- Write the image to file
- Format is guessed from extension
- ```
   bool imwrite(const string& file, 
                const Mat& img);
```
Write float images to *.exr files

- When storing floating point images OpenCV expects the values to be in $[0, 1]$ range
- When storing arbitrary values the values might be cut off
- Save to *.exr files to avoid this
- These files will store and read values as is without losing precision
#include <iostream>
#include <opencv2/opencv.hpp>
#include <string>

int main() {
    using Matf = cv::Mat_<float>;
    Matf image = Matf::zeros(10, 10);
    image.at<float>(5, 5) = 42.42f;
    std::string f = "test.exr";
    cv::imwrite(f, image);
    Matf copy = cv::imread(f, CV_LOAD_IMAGE_UNCHANGED);
    std::cout << copy.at<float>(5, 5) << std::endl;
    return 0;
}

Hint: try what happens when using png images instead
imshow

- Display the image on screen
- Needs a window to display the image
- `void imshow(const string& window_name, const Mat& mat)`

```cpp
#include <opencv2/opencv.hpp>

int main() {
    cv::Mat image = cv::imread("logo_opencv.png", CV_LOAD_IMAGE_COLOR);
    std::string window_name = "Window name";
    // Create a window.
    cv::namedWindow(window_name, cv::WINDOW_AUTOSIZE);
    cv::imshow(window_name, image); // Show image.
    cv::waitKey(); // Don't close window instantly.
    return 0;
}
```
OpenCV vector type

- OpenCV vector type: `cv::Vec<Type, SIZE>`
- Many typedefs available: `Vec3f`, `Vec3b`, etc.
- Used for pixels in multidimensional images:
  `mat.at<Vec3b>(row, col);`

```cpp
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace cv;

int main() {
    Mat mat = Mat::zeros(10, 10, CV_8UC3);
    std::cout << mat.at<Vec3b>(5, 5) << std::endl;
    Mat_<Vec3f> matf3 = Mat_<Vec3f>::zeros(10, 10);
    std::cout << matf3.at<Vec3f>(5, 5) << std::endl;
}
```
Mixing up types is painful!

- OpenCV trusts you to pick the type
- This can cause errors
- OpenCV interprets bytes stored in `cv::Mat` according to the type the user asks (similar to `reinterpret_cast`)
- Make sure you are using correct types!
Mixing up types is painful!

```cpp
#include <opencv2/opencv.hpp>

int main() {
    cv::Mat image = cv::Mat::zeros(800, 600, CV_8UC3);
    std::string window_name = "Window name";
    cv::namedWindow(window_name, cv::WINDOW_AUTOSIZE);
    cv::imshow(window_name, image);
    cv::waitKey();

    for (int r = 0; r < image.rows; ++r) {
        for (int c = 0; c < image.cols; ++c) {
            // WARNING! WRONG TYPE USED!
            image.at<float>(r, c) = 1.0f;
        }
    }
    cv::imshow(window_name, image);
    cv::waitKey();
    return 0;
}
```
SIFT Descriptors

- **SIFT**: Scale Invariant Feature Transform
- **Popular features**: illumination, rotation and translation invariant (to some degree)

(image courtesy of David G. Lowe)
SIFT Extraction With OpenCV

- SiftFeatureDetector to detect the keypoints
- SiftDescriptorExtractor to compute descriptors in keypoints

```cpp
// Detect keypoints.
SiftFeatureDetector detector;
vector<KeyPoint> keypoints;
detector.detect(input, keypoints);

// Show the keypoints on the image.
Mat image_with_keypoints;
drawKeypoints(input, keypoints, image_with_keypoints);

// Extract the SIFT descriptors.
SiftDescriptorExtractor extractor;
extractor.compute(input, keypoints, descriptors);
```
FLANN in OpenCV

- **FLANN**: Fast Library for Approximate Nearest Neighbors
- build K-d tree, search for neighbors there

```c++
// Create a kdtree for searching the data.
cv::flann::KDTreeIndexParams index_params;
cv::flann::Index kdtree(data, index_params);

// Search the nearest vector to some query
int k = 1;
Mat nearest_vector_idx(1, k, DataType<int>::type);
Mat nearest_vector_dist(1, k, DataType<float>::type);
kdtree.knnSearch(query, nearest_vector_idx, nearest_vector_dist, k);
```
OpenCV 2 with CMake

- **Install OpenCV 2 in the system**
  
  ```bash
  1. sudo add-apt-repository ppa:xqms/opencv-nonfree
  2. sudo apt update
  3. sudo apt install libopencv-dev libopencv-nonfree-dev
  ```

- **Find using** `find_package(OpenCV 2 REQUIRED)`
  
  ```bash
  1. find_package(OpenCV 2 REQUIRED)
  ```

- **Include** `OpenCV_INCLUDE_DIRS`
- **Link against** `OpenCV_LIBS`
  
  ```cpp
  1. add_library(some_lib some_lib_file.cpp)
  2. target_link_libraries(some_lib OpenCV_LIBS)
  3. add_executable(some_program some_file.cpp)
  4. target_link_libraries(some_program OpenCV_LIBS)
  ```
Additional OpenCV information

- We are using **OpenCV version 2**
- Running version 3 will lead to errors
- Example project with additional information about using SIFT and FLANN can be found here:
  
  https://gitlab.igg.uni-bonn.de/teaching/example_opencv
References

- **Macros:**

- **Lambda expressions:**

- **OpenCV SIFT:**
  - [https://docs.opencv.org/2.4/modules/nonfree/doc/feature_detection.html](https://docs.opencv.org/2.4/modules/nonfree/doc/feature_detection.html)