Modern C++ for Computer Vision and Image Processing

Igor Bogoslavskyi
Outline

Compilation flags and debugging

Functions

Header / Source Separation

Libraries

CMake Intro
Compilation flags

- There is a lot of flags that can be passed while compiling the code
- We have seen some already: 
  -std=c++11, -o, etc.

Other useful options:

- Enable all warnings, treat them as errors: 
  -Wall, -Wextra, -Werror
- Optimization options: 
  - -00 - no optimizations
  - -03 or -Ofast - full optimizations
- Keep debugging symbols: -g

Play with them with Compiler Explorer: https://godbolt.org/
Debugging tools

- The best option is to use `gdb`
- Insanely popular and powerful
- No build-in gui
- Use `gdbgui` for a user-friendly interface
- Install `gdbgui` from `pip`:
  ```
  sudo pip3 install --upgrade gdbgui
  ```

Tutorials, documentation and source code: https://gdbgui.com/
Functions

- 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
Good function example

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

vector<int> CreateVectorOfFullSquares(int size) {
    vector<int> result(size); // Vector of size `size`
    for (int i = 0; i < size; ++i) { result[i] = i * i; }
    return result;
}

int main() {
    auto squares = CreateVectorOfFullSquares(10);
    return 0;
}
```

- Is small enough to see all the code at once
- Name clearly states what the function does
- Function **does a single thing**
Bad function example

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

vector<int> Func(int a, bool b) {
    if (b) { return vector<int>(10, a); }
    vector<int> vec(a);
    for (int i = 0; i < a; ++i) { vec[i] = a * i; }
    if (vec.size() > a * 2) { vec[a] /= 2.0f; }
    return vec;
}
```

- Name of the function means nothing
- Names of variables mean nothing
- Function does not have a single purpose
Function declaration can be separated from the implementation details

Function **declaration** sets up an interface

```c
void FuncName(int param);
```

Function **definition** holds the implementation of the function that can even be hidden from the user

```c
void FuncName(int param) {
    // Implementation details.
    cout << "This function is called FuncName! ";
    cout << "Did you expect anything useful from it?";
}
```
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

```cpp
1  void DoSmth(std::string huge_string); // Slow.
2  void DoSmth(std::string& huge_string); // Faster.
```

Is the string still the same?

```cpp
1  string hello = "some_important_long_string";
2  DoSmth(hello);
```

**Unknown** without looking into `DoSmth()`!
Pass by reference intuition

- **Pass by reference:**
  - `void fillCup(Cup &cup);`
  - `cup` is full

- **Pass by value:**
  - `void fillCup(Cup cup);`
  - A copy of `cup` is full
  - `cup` is still empty
Solution: use const references

- Pass `const` reference to the function
- Great speed as we pass a reference
- Passed object stays intact

```cpp
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
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

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

string Func(int num) { return "int"; }
string Func(const string& str) { return "string"; }

int main() {
    cout << Func(1) << endl;
    cout << Func("hello") << endl;
    return 0;
}
```

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

Google style: https://google.github.io/styleguide/cppguide.html#Default_Arguments
Example: default arguments

```cpp
#include <iostream> // std::cout, std::endl
using namespace std;

string SayHello(const string& to_whom = "world") {
    return "Hello " + to_whom + "!";
}

int main() {
    cout << SayHello() << endl;
    cout << SayHello("students") << endl;
    return 0;
}
```
Don’t reinvent the wheel

- When using `std::vector`, `std::array`, etc. try to avoid writing your own functions.
- Use `#include <algorithm>`
- There is a lot of functions in `std` which are at least as fast as hand-written ones:

```cpp
std::vector<float> v;
// Filling the vector omitted here.
std::sort(v.begin(), v.end()); // Sort ascending.
float sum = std::accumulate(v.begin(), v.end(), 0.0f);
float product = std::accumulate(v.begin(), v.end(), 1.0f, std::multiplies<float>());
```

Header / Source Separation

- Move all declarations to header files (*.h)
- Implementation goes to *.cpp or *.cc

```cpp
// some_file.h
Type SomeFunc(... args ...);

// some_file.cpp
#include "some_file.h"
Type SomeFunc(... args ...) { /* code */ }

// program.cpp
#include "some_file.h"
int main() {
    SomeFunc(/* args */);
    return 0;
}
```
How to build this?

1. folder/
   - tools.h
   - tools.cpp
   - main.cpp

**Short:** we separate the code into modules

**Declaration:** tools.h

```cpp
#pragma once // Ensure file is included only once

void MakeItSunny();
void MakeItRain();
```
How to build this?

**Definition:** tools.cpp

```cpp
#include <iostream>
#include "tools.h"

void MakeItRain() {
    // important weather manipulation code
    std::cout << "Here! Now it rains! Happy?\n";
}

void MakeItSunny() { std::cerr << "Not available\n"; }
```

**Calling:** main.cpp

```cpp
#include "tools.h"

int main() {
    MakeItRain();
    MakeItSunny();
    return 0;
}
```
Just build it as before?

c++ -std=c++11 main.cpp -o main

Error:

1 /tmp/tools_main-0eacf5.o: In function `main':
2 tools_main.cpp: undefined reference to `makeItRain()'
3 tools_main.cpp: undefined reference to `makeItSunny()'
4 clang: error: linker command failed with exit code 1
5 (use -v to see invocation)
Use modules and libraries!

Compile modules:
```
c++ -std=c++11 -c tools.cpp -o tools.o
```

Organize modules into libraries:
```
ar rcs libtools.a tools.o <other_modules>
```

Link libraries when building code:
```
c++ -std=c++11 main.cpp -L . -ltools -o main
```

Run the code:
```
./main
```
Libraries

- **Library**: multiple object files that are logically connected

- Types of libraries:
  - **Static**: faster, take a lot of space, become part of the end binary, named: `lib*.a`
  - **Dynamic**: slower, can be copied, referenced by a program, named `lib*.so`

- Create a static library with
  ```
  ar rcs libname.a module.o module.o ...
  ```

- Static libraries are just archives just like `zip/tar/...`
What is linking?

- The library is a binary object that contains the **compiled implementation** of some methods.
- Linking maps a function declaration to its compiled implementation.
- To use a library we need a header and the compiled library object.
Use CMake to simplify the build

- One of the most popular build tools
- Does not build the code, generates files to feed into a build system
- Cross-platform
- Very powerful, still build receipt is readable
- The library creation and linking can be rewritten as follows:

```
1 add_library(tools tools.cpp)
2 add_executable(main main.cpp)
3 target_link_libraries(main tools)
```
Typical project structure

- project_name/
  - CMakeLists.txt
  - build/ # All generated build files
  - bin/
    - tools_demo
  - lib/
    - libtools.a
  - src/
    - CMakeLists.txt
    - project_name
      - CMakeLists.txt
      - tools.h
      - tools.cpp
      - tools_demo.cpp
  - tests/ # Tests for your code
    - test_tools.cpp
  - readme.md # How to use your code
Build process

- **CMakeLists.txt** defines the whole build
- **CMake reads CMakeLists.txt sequentially**
- **Build process:**
  1. cd <project_folder>
  2. mkdir build
  3. cd build
  4. cmake ..
  5. make -j2  # pass your number of cores here
First working CMakeLists.txt

```cmake
project(first_project) # Mandatory.
cmake_minimum_required(VERSION 3.1) # Mandatory.
set(CMAKE_CXX_STANDARD 11) # Use c++11.
# tell cmake to output binaries here:
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
# tell cmake where to look for *.h files
include_directories(${PROJECT_SOURCE_DIR}/src)
# create library "libtools"
add_library(tools src/tools.cpp)
# add executable main
add_executable(main src/tools_main.cpp)
# tell the linker to bind these objects together
target_link_libraries(main tools)
```
Useful commands in CMake

- Just a scripting language
- Has features of a scripting language, i.e. functions, control structures, variables, etc.
- All variables are string
- Set variables with `set(VAR VALUE)`
- Get value of a variable with `${VAR}`
- Show a message `message(STATUS "message")`
- Also possible `WARNING, FATAL_ERROR`
Use CMake in your builds

- Build process is standard and simple
- No need to remember sequence of commands
- All generated build files are in one place
- CMake detects changes to the files
- Do this after changing files:
  1. `cd project/build`
  2. `make -j2` # pass your number of cores here
Set compilation options in CMake

```
1 set (CMAKE_CXX_STANDARD 14)
2 # Set build type if not set.
3 if(NOT CMAKE_BUILD_TYPE)
4  set(CMAKE_BUILD_TYPE Release)
5 endif()
6 # Set additional flags.
7 set(CMAKE_CXX_FLAGS "-Wall -Wextra")
8 set(CMAKE_CXX_FLAGS_DEBUG "-g -00")
9 set(CMAKE_CXX_FLAGS_RELEASE "-03")
```

- `-Wall -Wextra`: show all warnings
- `-g`: keep debug information in binary
- `-0<num>`: optimization level in `{0, 1, 2, 3}`
  - 0: no optimization
  - 3: full optimization
Remove build folder for performing a clean build

- Sometimes you want a clean build
- It is very easy to do with CMake
  1. cd project/build
  2. make clean [remove generated binaries]
  3. rm -r * [make sure you are in build folder]
Use pre-compiled library

- Sometimes you get a compiled library
- You can use it in your build
- For example, given `libtools.so` it can be used in the project as follows:

```cpp
1 find_library(TOOLS
2     NAMES tools
3     PATHS ${LIBRARY_OUTPUT_PATH})
4 # Use it for linking:
5 target_link_libraries(<some_binary> ${TOOLS})
```
References

- **Compiler Explorer:**
  https://godbolt.org/

- **Gdbgui:**
  https://gdbgui.com/

- **Gdbgui tutorial:**
  https://www.youtube.com/watch?v=em842geJhfk

- **CMake website:**
  https://cmake.org/

- **Modern CMake Tutorial:**
  https://www.youtube.com/watch?v=eC9-iRN2b04