

Modern C++ for Computer Vision

Lecture 5: I/O Files, Classes

Ignacio Vizzo, Rodrigo Marcuzzi, Cyrill Stachniss

C++ Utilities

C++ includes a variety of utility libraries that provide functionality ranging from bit-counting to partial function application.

These libraries can be broadly divided into two groups:

- language support libraries.
- general-purpose libraries.

Language support

Provide classes and functions that interact closely with language features and support common language idioms.

- Type support(`std::size_t`).
- Dynamic memory management(`std::shared_ptr`).
- Error handling(`std::exception`, `assert`).
- Initializer list(`std::vector{1, 2}`).
- Much more...

General-purpose Utilities

- Program utilities(`std::abort`).
- Date and Time(`std::chrono::duration`).
- Optional, variant and any(`std::variant`).
- Pairs and tuples(`std::tuple`).
- Swap, forward and move(`std::move`).
- Hash support(`std::hash`).
- Formatting library(coming in C++20).
- Much more...

std::swap

```
1 int main() {  
2     int a = 3;  
3     int b = 5;  
4  
5     // before  
6     std::cout << a << ' ' << b << '\n';  
7  
8     std::swap(a, b);  
9  
10    // after  
11    std::cout << a << ' ' << b << '\n';  
12 }
```

Output:

```
1 3 5  
2 5 3
```

std::variant

```
1 int main() {
2     std::variant<int, float> v1;
3     v1 = 12;    // v contains int
4     cout << std::get<int>(v1) << endl;
5     std::variant<int, float> v2{3.14F};
6     cout << std::get<1>(v2) << endl;
7
8     v2 = std::get<int>(v1);    // assigns v1 to v2
9     v2 = std::get<0>(v1);    // same as previous line
10    v2 = v1;                  // same as previous line
11    cout << std::get<int>(v2) << endl;
12 }
```

Output:

```
1 12
2 3.14
3 12
```

std::any

```
1 int main() {
2     std::any a;    // any type
3
4     a = 1;    // int
5     cout << any_cast<int>(a) << endl;
6
7     a = 3.14;    // double
8     cout << any_cast<double>(a) << endl;
9
10    a = true;    // bool
11    cout << std::boolalpha << any_cast<bool>(a) << endl;
12 }
```

Output:

```
1 1
2 3.14
3 true
```

std::optional

```
1 std::optional<std::string> StringFactory(bool create) {
2     if (create) {
3         return "Modern C++ is Awesome";
4     }
5     return {};
6 }
7
8 int main() {
9     cout << StringFactory(true).value() << '\n';
10    cout << StringFactory(false).value_or(":(") << '\n';
11 }
```

Output:

```
1 Modern C++ is Awesome
2 :(
```


std::tuple

```
1 int main() {
2     std::tuple<double, char, string> student1;
3     using Student = std::tuple<double, char, string>;
4     Student student2{1.4, 'A', "Jose"};
5     PrintStudent(student2);
6     cout << std::get<string>(student2) << endl;
7     cout << std::get<2>(student2) << endl;
8
9     // C++17 structured binding:
10    auto [gpa, grade, name] = make_tuple(4.4, 'B', "");
11 }
```

Output:

```
1 GPA: 1.4, grade: A, name: Jose
2 Jose
3 Jose
```

std::chrono

```
1 #include <chrono>
2
3 int main() {
4     auto start = std::chrono::steady_clock::now();
5     cout << "f(42) = " << fibonacci(42) << '\n';
6     auto end = chrono::steady_clock::now();
7
8     chrono::duration<double> sec = end - start;
9     cout << "elapsed time: " << sec.count() << "s\n";
10 }
```

Output:

```
1 f(42) = 267914296
2 elapsed time: 1.84088s
```

Much more utilites

Just spend some time looking around:

- <https://en.cppreference.com/w/cpp/utility>

Copying big objects

```
1 // MyClass has a private member with a lot of strings
2 struct MyClass {
3     int id_ = 0;
4     std::vector<std::string> names_{
5         "name", "name", "name", "name", "name", "name", "name", "name", "name"};
6 };
```

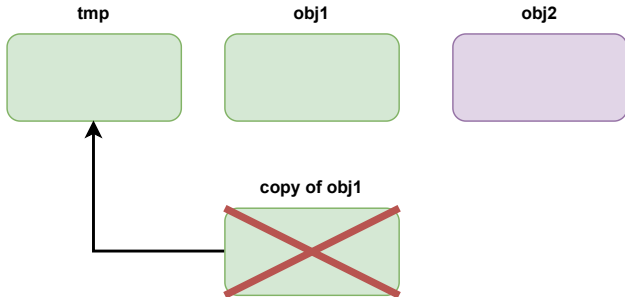
Copying big objects

```
1 // MyClass has a private member with a lot of strings
2 struct MyClass {
3     int id_ = 0;
4     std::vector<std::string> names_{
5         "name", "name", "name", "name", "name", "name", "name", "name", "name"};
6 };
7
8 void copy_swap(MyClass& obj1, MyClass& obj2) {
9     MyClass tmp = obj1; // copy obj1 to tmp
10    obj1 = obj2;         // copy obj2 to obj1
11    obj2 = tmp;          // copy tmp to obj1
12 }
```

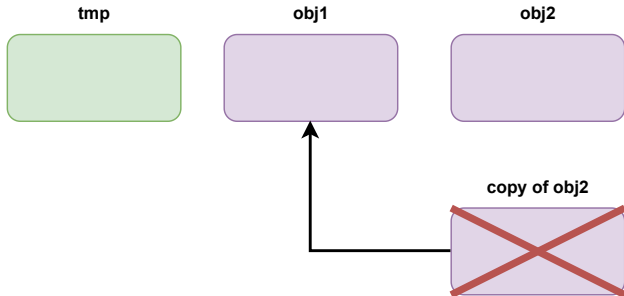
Copying big objects



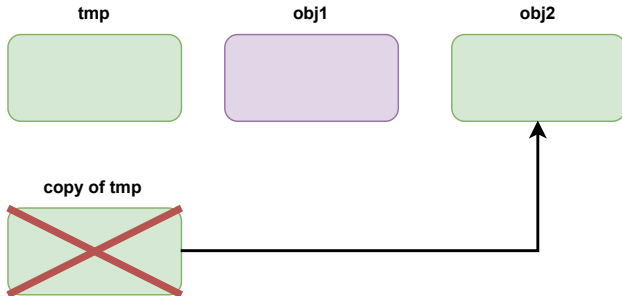
Copying big objects



Copying big objects



Copying big objects



Copying big objects: using move

```
1 // MyClass has a private member with a lot of strings
2 struct MyClass {
3     int id_ = 0;
4     std::vector<std::string> names_{
5         "name", "name", "name", "name", "name", "name", "name", "name", "name", "name"};
6 };
7
8 void copy_swap(MyClass& obj1, MyClass& obj2) {
9     MyClass tmp = obj1; // copy obj1 to tmp
10    obj1 = obj2;         // copy obj2 to obj1
11    obj2 = tmp;          // copy tmp to obj1
12 }
13
14 void move_swap(MyClass& obj1, MyClass& obj2) {
15     MyClass tmp = std::move(obj1); // move obj1 to tmp
16     obj1 = std::move(obj2);         // move obj2 to obj1
17     obj2 = std::move(tmp);          // move tmp to obj1
18 }
```

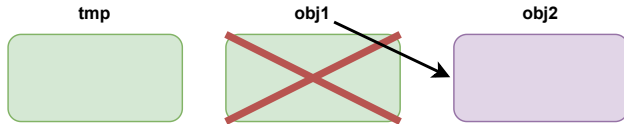
Copying big objects: using move



Copying big objects: using move



Copying big objects: using move



Copying big objects: using move



Intuition lvalues, rvalues

- Every expression is an **lvalue** or an **rvalue**
- **lvalues** can be written on the **left** of assignment operator (=)
- **rvalues** are all the other expressions
- Explicit rvalue defined using **&&**
- Use **std::move(...)** to explicitly convert an lvalue to an rvalue

```
1 int a;           // "a" is an lvalue
2 int& a_ref = a;   // "a" is an lvalue
3                 // "a_ref" is a reference to an lvalue
4 a = 2 + 2;        // "a" is an lvalue,
5                 // "2 + 2" is an rvalue
6 int b = a + 2;     // "b" is an lvalue,
7                 // "a + 2" is an rvalue
8 int&& c = std::move(a); // "c" is an rvalue
```

Important `std::move`

- The `std::move()` is a standard-library function returning an `rvalue` reference to its argument.
- `std::move(x)` means “give me an `rvalue` reference to `x`.”
- That is, `std::move(x)` **does not move anything**;
- Instead, it allows a user to move `x`.

Never access values after move

The value after `move` is undefined

```
1 string str = "Hello";
2 vector<string> v;
3
4 // uses the push_back(const T&) overload, which means
5 // we'll incur the cost of copying str
6 v.push_back(str);
7 cout << "After copy, str is " << str << endl;
8
9 // uses the rvalue reference push_back(T&&) overload,
10 // which means no strings will be copied; instead,
11 // the contents of str will be moved into the vector.
12 // This is less expensive, but also means str might
13 // now be empty.
14 v.push_back(move(str));
15 cout << "After move, str is " << str << endl;
```



How to think about `std::move`

- Think about **ownership**
- Entity **owns** a variable if it deletes it, e.g.
 - A function scope owns a variable defined in it
 - An object of a class owns its data members
- **Moving a variable transfers ownership** of its resources to another variable
- When designing your program think **“who should own this thing?”**
- **Runtime:** better than copying, worse than passing by reference

Error handling with exceptions

- We can **"throw"** an exception if there is an error
- STL defines classes that represent exceptions. Base class: `std::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:

```
1 // if there is an error
2 if (badEvent) {
3     string msg = "specific error string";
4     // throw error
5     throw runtime_error(msg);
6 }
7 ... some cool code if all ok ...
```

Logic Error: an error in logic of the user

```
1 throw logic_error(msg);
```

catch exceptions

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

```
1 try {
2     // some code that can throw exceptions z.B.
3     x = someUnsafeFunction(a, b, c);
4 }
5 // we can catch multiple types of exceptions
6 catch ( runtime_error &ex ) {
7     cerr << "Runtime error: " << ex.what() << endl;
8 } catch ( logic_error &ex ) {
9     cerr << "Logic error: " << ex.what() << endl;
10 } catch ( exception &ex ) {
11     cerr << "Some exception: " << ex.what() << endl;
12 } catch ( ... ) { // all others
13     cerr << "Error: unknown exception" << endl;
14 }
```

Intuition

- Only used for **“exceptional behavior”**
- **Often misused**: e.g. wrong parameter should not lead to an exception
- **GOOGLE-STYLE** Don't use exceptions
- <https://en.cppreference.com/w/cpp/error>

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
5 // ifstream: stream for input from file
6 std::ifstream f_in(string& file_name, Mode mode);
7
8 // ofstream: stream for output to file
9 std::ofstream f_out(string& file_name, Mode mode);
10
11 // stream for input and output to file
12 std::fstream f_in_out(string& file_name, Mode mode);
```

There are many modes under which a file can be opened

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

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        cout << i << ", " << a << ", "
14             << s << ", " << b << endl;
15    }
16    return (0);
17 }
```

Regular columns

Use it when:

- The file contains 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 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 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/ivizzo/.bashrc  
5 =====  
6 2.34  
7 1 2.23  
8 ER SIE ES
```

```
1 #include <fstream> // For the file streams.
2 #include <iostream>
3 using namespace std;
4 int main() {
5     string line, file_name;
6     ifstream input("test_bel.txt", ios_base::in);
7     // Read data line-wise.
8     while (getline(input, line)) {
9         cout << "Read: " << line << endl;
10        // String has a find method.
11        string::size_type loc = line.find("filename", 0);
12        if (loc != string::npos) {
13            file_name = line.substr(line.find("=", 0) + 1,
14                                   string::npos);
15        }
16    }
17    cout << "Filename found: " << file_name << endl;
18    return (0);
19 }
```

Writing into text files

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

```
1 #include <iomanip> // For setprecision.
2 #include <fstream>
3 using namespace std;
4 int main() {
5     string filename = "out.txt";
6     ofstream outfile(filename);
7     if (!outfile.is_open()) { return EXIT_FAILURE; }
8     double a = 1.123123123;
9     outfile << "Just string" << endl;
10    outfile << setprecision(20) << a << endl;
11    return 0;
12 }
```

Writing to binary files

- We write a **sequence of bytes**
- We must document the structure well, otherwise none can read the file
- Writing/reading is **fast**
- No precision loss for floating point types
- Substantially **smaller** than `ascii`-files
- **Syntax**

```
1 file.write(reinterpret_cast<char*>(&a), sizeof(a));
```

Writing to binary files

```
1 #include <fstream> // for the file streams
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 int main() {
7     string file_name = "image.dat";
8     ofstream file(file_name, ios_base::out | ios_base::binary);
9     int rows = 2;
10    int cols = 3;
11    vector<float> vec(rows * cols);
12    fill(vec.begin(), vec.end(), -100.001);
13    file.write(reinterpret_cast<char*>(&rows), sizeof(rows));
14    file.write(reinterpret_cast<char*>(&cols), sizeof(cols));
15    file.write(reinterpret_cast<char*>(&vec.front()),
16               vec.size() * sizeof(float));
17    return 0;
18 }
```

Writing to ascii files

```
1 #include <fstream> // for the file streams
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 int main() {
7     string file_name = "image.txt";
8     ofstream file;
9     file.open(file_name);
10    int rows = 2;
11    int cols = 3;
12    vector<float> vec(rows * cols);
13    fill(vec.begin(), vec.end(), -100.001);
14    file << rows << "\n";
15    file << cols << "\n";
16    for(int i=0; i< vec.size(); i++){
17        file << vec[i] << "\n";
18    }
19    file.close();
20    return 0;
21 }
```


Reading from binary files

- We read a **sequence of bytes**
- Binary files are not human-readable
- We must know the structure of the contents
- **Syntax**

```
1 file.read(reinterpret_cast<char*>(&a), sizeof(a));
```

Reading from binary files

```
1 #include <fstream>
2 #include <iostream>
3 #include <vector>
4 using namespace std;
5 int main() {
6     string file_name = "image.dat";
7     int r = 0, c = 0;
8     ifstream in(file_name,
9                 ios_base::in | ios_base::binary);
10    if (!in) { return EXIT_FAILURE; }
11    in.read(reinterpret_cast<char*>(&r), sizeof(r));
12    in.read(reinterpret_cast<char*>(&c), sizeof(c));
13    cout << "Dim: " << r << " x " << c << endl;
14    vector<float> data(r * c, 0);
15    in.read(reinterpret_cast<char*>(&data.front()),
16            data.size() * sizeof(data.front()));
17    for (float d : data) { cout << d << endl; }
18    return 0;
19 }
```

Important facts

Pros

- I/O Binary files is **faster** than ASCII format.
- Size of files is **drastically** smaller.
- There are many libraries to facilitate **serialization**.

Cons

- Hard to read Syntax.
- File is not readable by human.
- You need to know the format before reading.

Saving binaries: the wrong way

- Save data and format separately
- Have to keep track of two files

point_cloud.bin

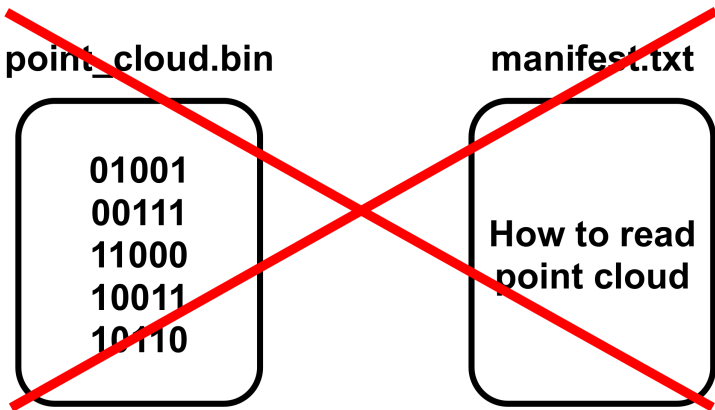
**01001
00111
11000
10011
10110**

manifest.txt

**How to read
point cloud**

Saving binaries: the wrong way

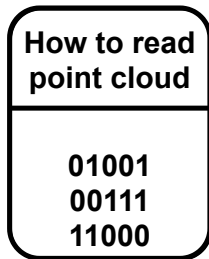
- Save data and format separately
- Have to keep track of two files



Saving binaries: the right way

- Save data and format in the same file
- Self contained in a single file

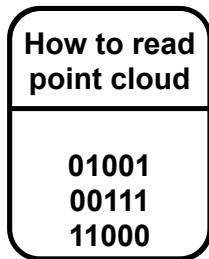
point_cloud.ply



Saving binaries: the right way

- Save data and format in the same file
- Self contained in a single file

point_cloud.ply



- Don't create you own format, use standards

C++17 Filesystem library

- Introduced in C++17.
- Use to perform operations on:
 - paths
 - regular files
 - directories
- Inspired in `boost::filesystem`
- Makes your life easier.
- <https://en.cppreference.com/w/cpp/filesystem>

directory_iterator

```
1 #include <filesystem>
2 namespace fs = std::filesystem;
3
4 int main() {
5     fs::create_directories("sandbox/a");
6     std::ofstream("sandbox/file1.txt");
7     std::ofstream("sandbox/file2.txt");
8     for (auto& p : fs::directory_iterator("sandbox")) {
9         std::cout << p.path() << '\n';
10    }
11    fs::remove_all("sandbox");
12 }
```

Output:

```
1 "sandbox/a"
2 "sandbox/file1.txt"
3 "sandbox/file2.txt"
```

filename_part1

```
1 #include <filesystem>
2 namespace fs = std::filesystem;
3
4 int main() {
5     cout << fs::path("/foo/bar.txt").filename() << '\n'
6         << fs::path("/foo/.bar").filename() << '\n'
7         << fs::path("/foo/bar/").filename() << '\n'
8         << fs::path("/foo/.").filename() << '\n'
9         << fs::path("/foo/..").filename() << '\n';
10 }
```

Output:

```
1 "bar.txt"
2 ".bar"
3 ""
4 "."
5 ".."
```

filename_part2

```
1 #include <filesystem>
2 namespace fs = std::filesystem;
3
4 int main() {
5     cout << fs::path("/foo/.bar").filename() << '\n'
6         << fs::path(".").filename() << '\n'
7         << fs::path("..").filename() << '\n'
8         << fs::path("/").filename() << '\n'
9         << fs::path("//host").filename() << '\n';
10 }
```

Output:

```
1 ".bar"
2 "."
3 ".."
4 ""
5 "host"
```

extension_part1

```
1 #include <filesystem>
2 namespace fs = std::filesystem;
3
4 int main() {
5     cout << fs::path("/foo/bar.txt").extension() << '\n'
6         << fs::path("/foo/bar.").extension() << '\n'
7         << fs::path("/foo/bar").extension() << '\n'
8         << fs::path("/foo/bar.png").extension() << '\n';
9 }
```

Output:

```
1 ".txt"
2 "."
3 ""
4 ".png"
```

extension_part2

```
1 #include <filesystem>
2 namespace fs = std::filesystem;
3
4 int main() {
5     cout << fs::path("/foo/.").extension() << '\n'
6         << fs::path("/foo/..").extension() << '\n'
7         << fs::path("/foo/.hidden").extension() << '\n'
8         << fs::path("/foo/..bar").extension() << '\n';
9 }
```

Output:

```
1 ""
2 ""
3 ""
4 ".bar"
```

stem

```
1 #include <filesystem>
2 namespace fs = std::filesystem;
3
4 int main() {
5     cout << fs::path("/foo/bar.txt").stem() << endl
6         << fs::path("/foo/00000.png").stem() << endl
7         << fs::path("/foo/.bar").stem() << endl;
8 }
```

Output:

```
1 "bar"
2 "00000"
3 ".bar"
```

exists

```
1 void demo_exists(const fs::path& p) {
2     cout << p;
3     if (fs::exists(p))    cout << " exists\n";
4     else                  cout << " does not exist\n";
5 }
6
7 int main() {
8     fs::create_directory("sandbox");
9     ofstream("sandbox/file"); // create regular file
10    demo_exists("sandbox/file");
11    demo_exists("sandbox/cacho");
12    fs::remove_all("sandbox");
13 }
```

Output:

```
1 "sandbox/file" exists
2 "sandbox/cacho" does not exist
```

Must Watch

Bag of Visual Words Introduction



<https://youtu.be/a4cFONdc6nc>

Suggested Video

Features Descriptors



<https://youtu.be/CMolhcwtGAU>

References

- **Utility Library**

<https://en.cppreference.com/w/cpp/utility>

- **Error handling**

<https://en.cppreference.com/w/cpp/error>

- **IO Library**

<https://en.cppreference.com/w/cpp/io>

- **Filesystem Library**

<https://en.cppreference.com/w/cpp/filesystem>