Modern C++ for Computer Vision and Image Processing

Lecture 5: I/O Files, Classes

Ignacio Vizzo and Cyrill Stachniss

UNIVERSITÄT BONN

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

```cpp
int main() {
  int a = 3;
  int b = 5;

  // before
  std::cout << a << ' ' << b << '\n';

  std::swap(a, b);

  // after
  std::cout << a << ' ' << b << '\n';
}
```

Output:

```
3 5
5 3
```

std::variant

```
int main() {
  std::variant<int, float> v1;
  v1 = 12;  // v contains int
  cout << std::get<int>(v1) << endl;
  std::variant<int, float> v2{3.14F};
  cout << std::get<1>(v2) << endl;

  v2 = std::get<int>(v1);  // assigns v1 to v2
  v2 = std::get<0>(v1);    // same as previous line
  v2 = v1;                 // same as previous line
  cout << std::get<int>(v2) << endl;
}
```

Output:

```
12
3.14
12
```

std::any

```cpp
int main() {
  std::any a;  // any type

  a = 1;  // int
  cout << any_cast<int>(a) << endl;

  a = 3.14;  // double
  cout << any_cast<double>(a) << endl;

  a = true;  // bool
  cout << std::boolalpha << any_cast<bool>(a) << endl;
}
```

Output:

```
1
3.14
true
```

std::optional

```cpp
std::optional<std::string> StringFactory(bool create) {
  if (create) {
    return "Modern C++ is Awesome";
  }
  return {};
}

int main() {
  cout << StringFactory(true).value() << '\n';
  cout << StringFactory(false).value_or(":(") << '\n';
}
```

Output:

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

std::tuple

```
int main() {
  std::tuple<double, char, string> student1;
  using Student = std::tuple<double, char, string>;
  Student student2{1.4, 'A', "Jose"};
  PrintStudent(student2);
  cout << std::get<string>(student2) << endl;
  cout << std::get<2>(student2) << endl;

  // C++17 structured binding:
  auto [gpa, grade, name] = make_tuple(4.4, 'B', "");
}
```

Output:

```
GPA: 1.4, grade: A, name: Jose
Jose
Jose
```

std::chrono

```cpp
#include <chrono>

int main() {
  auto start = std::chrono::steady_clock::now();
  cout << "f(42) = " << fibonacci(42) << '\n';
  auto end = chrono::steady_clock::now();

  chrono::duration<double> sec = end - start;
  cout << "elapsed time: " << sec.count() << "s\n";
}
```

Output:

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

Much more utilites

Just spend some time looking around:

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

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

```
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
- https://en.cppreference.com/w/cpp/error

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

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

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 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 }
```

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
```

```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

```
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  using namespace std;
4
5  int main() {
6    string file_name = "image.dat";
7    ofstream file(file_name, ios_base::out | ios_base::binary);
8    int rows = 2;
9    int cols = 3;
10   vector<float> vec(rows * cols);
11   file.write(reinterpret_cast<char*>(&rows), sizeof(rows));
12   file.write(reinterpret_cast<char*>(&cols), sizeof(cols));
13   file.write(reinterpret_cast<char*>(&vec.front()),
14                                      vec.size() * sizeof(float));
15   return 0;
16 }
```

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

- Ugly Syntax.
- File is not readable by human.
- You need to now the format before reading.
- You need to use this for your homeworks.

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

```cpp
#include <filesystem>
namespace fs = std::filesystem;

int main() {
  fs::create_directories("sandbox/a/b");
  std::ofstream("sandbox/file1.txt");
  std::ofstream("sandbox/file2.txt");
  for (auto& p : fs::directory_iterator("sandbox")) {
    std::cout << p.path() << '\n';
  }
  fs::remove_all("sandbox");
}
```

Output:

```
"sandbox/a"
"sandbox/file1.txt"
"sandbox/file2.txt"
```

filename_part1

```cpp
#include <filesystem>
namespace fs = std::filesystem;

int main() {
  cout << fs::path("/foo/bar.txt").filename() << '\n'
       << fs::path("/foo/.bar").filename() << '\n'
       << fs::path("/foo/bar/").filename() << '\n'
       << fs::path("/foo/.").filename() << '\n'
       << fs::path("/foo/..").filename() << '\n';
}
```

Output:

```
"bar.txt"
".bar"
""
"."
".."
```

filename_part2

```cpp
#include <filesystem>
namespace fs = std::filesystem;

int main() {
  cout << fs::path("/foo/.bar").filename() << '\n'
       << fs::path(".").filename() << '\n'
       << fs::path("..").filename() << '\n'
       << fs::path("/").filename() << '\n'
       << fs::path("//host").filename() << '\n';
}
```

Output:

```
".bar"
"."
".."
""
"host"
```

extension_part1

```cpp
#include <filesystem>
namespace fs = std::filesystem;

int main() {
  cout << fs::path("/foo/bar.txt").extension() << '\n'
       << fs::path("/foo/bar.").extension() << '\n'
       << fs::path("/foo/bar").extension() << '\n'
       << fs::path("/foo/bar.png").extension() << '\n';
}
```

Output:

```
".txt"
"."
""
".png"
```

extension_part2

```cpp
#include <filesystem>
namespace fs = std::filesystem;

int main() {
  cout << fs::path("/foo/.").extension() << '\n'
       << fs::path("/foo/..").extension() << '\n'
       << fs::path("/foo/.hidden").extension() << '\n'
       << fs::path("/foo/..bar").extension() << '\n';
}
```

Output:

```
""
""
""
".bar"
```

stem

```cpp
#include <filesystem>
namespace fs = std::filesystem;

int main() {
  cout << fs::path("/foo/bar.txt").stem() << endl
       << fs::path("/foo/00000.png").stem() << endl
       << fs::path("/foo/.bar").stem() << endl;
}
```

Output:

```
"bar"
"00000"
".bar"
```

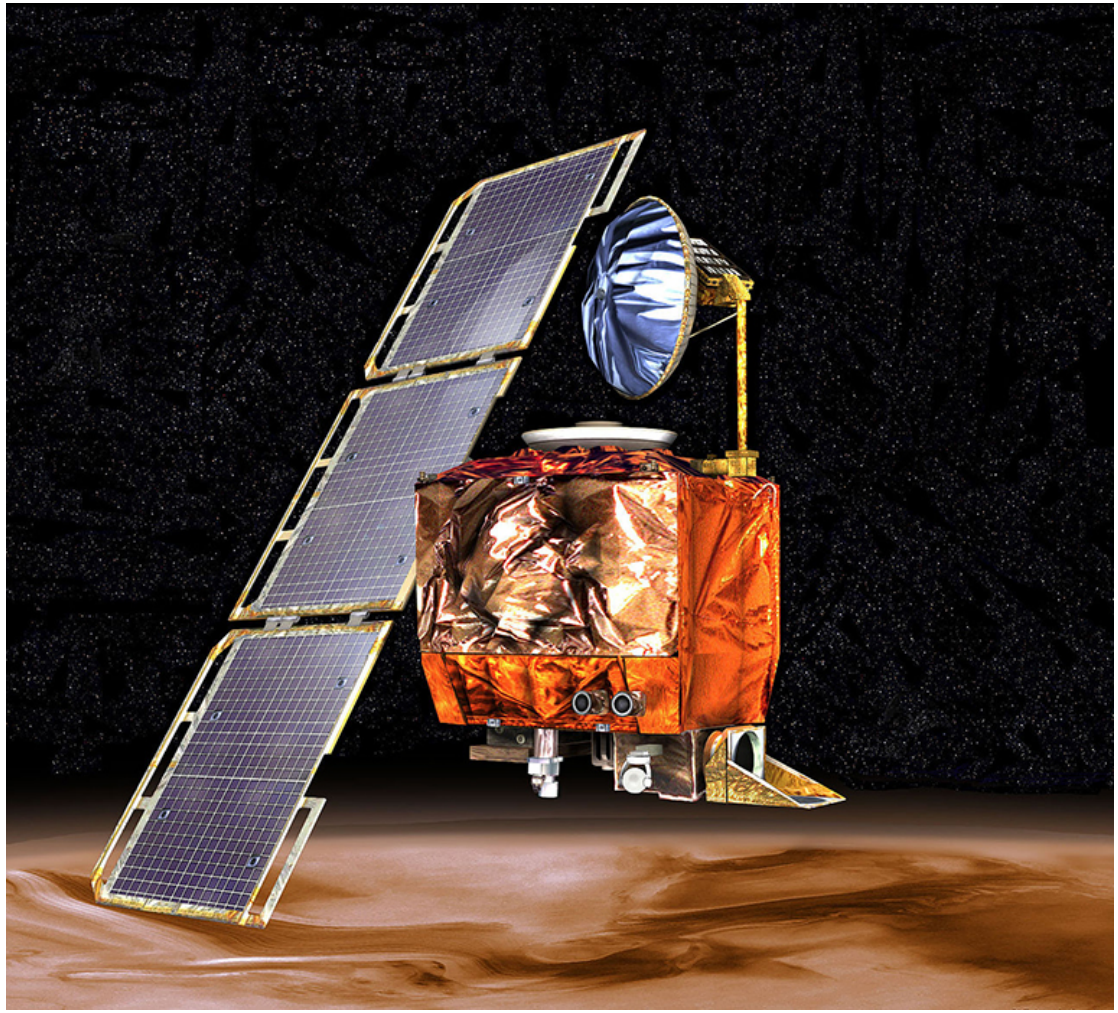
exists

```
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
```

Types are indeed important



https://www.simscale.com/blog/2017/12/nasa-mars-climate-orbiter-metric/

Type safety

bad – the unit is ambiguous

```
1 void blink_led_bad(int time_to_blink) {
2   // do something with time_to_blink
3 }
```

- What if I call `blink_led_bad()` with wrong units?
- When I will detect the error?

good – the unit is explicit

```
1 void blink_led_good(miliseconds time_to_blink) {
2   // do something with time_to_blink
3 }
```

Example taken from: https://youtu.be/fX2W3nNjJIo

Type safety

good – the unit is explicit

```
1 void blink_led_good(miliseconds time_to_blink) {
2   // do something with time_to_blink
3 }
```

Usage

```
1 void use() {
2   blink_led_good(100);     // ERROR: What unit?
3   blink_led_good(100ms);   //
4   blink_led_good(5s);      // ERROR: Bad unit
5 }
```

Example taken from: https://youtu.be/fX2W3nNjJIo

Want more flexibilty?

```
1  template <class rep, class period>
2  void blink_led(duration<rep, period> blink_time) {
3    // millisecond is the smallest relevant unit
4    auto x_ms = duration_cast<miliseconds>(blink_time);
5    // do something else with x_ms
6  }
7
8  void use() {
9    blink_led(2s);      // Works fine
10   blink_led(150ms);   // Also, works fine
11   blink_led(150);     // ERROR, which unit?
12 }
```

Example taken from: https://youtu.be/fX2W3nNjJIo

Type safety in our field

BAD Example: ROS 1

```
1 // ...
2 //
3 // %Tag(LOOP_RATE)%
4 ros::Rate loop_rate(10);
5 // %EndTag(LOOP_RATE)%
6 //
7 // ...
```

`loop_rate` in which units? `Hz`, `ms` ???

https://github.com/ros/ros_tutorials/blob/noetic-devel/roscpp_tutorials/talker/talker.cpp

Type safety in our field

GOOD Example: ROS 2

```
1 //  ...
2 //
3 timer_ = create_wall_timer(100ms, timer_callback);
4 //
5 //  ...
```

- Same functionality as previous example
- Better code, better readability
- Safer
- Guaranteed to run every 100ms(10 Hz)

https://github.com/ros2/examples/blob/master/rclcpp/topics/minimal_publisher/lambda.cpp

Class Basics

"C++ classes are a **tools** for creating **new types** that can be used as conveniently as the built-in types. In addition, derived classes and templates allow the programmer to express **relationships** among classes and to take advantage of such relationships."

Extract from: Section 16 of "The C++ Programming Language Book by Bjarne Stroustrup"

Class Basics

"A type is a concrete representation of a **concept** (an idea, a notion, etc.). A program that provides types that closely match the concepts of the application tends to be easier to **understand**, easier to **reason** about, and easier to **modify** than a program that does not."

Extract from: Section 16 of "The C++ Programming Language Book by Bjarne Stroustrup"

Class Basics

- A `class` is a user-defined type
- A `class` consists of a set of members. The most common kinds of members are data members and member functions
- Member functions can define the meaning of initialization (creation), copy, move, and cleanup (destruction)
- Members are accessed using `.(dot)` for objects and `->` `(arrow)` for pointers

Extract from: Section 16 of "The C++ Programming Language Book by Bjarne Stroustrup"

Class Basics

- Operators, such as `+`, `!`, and `[]`, can be defined for a `class`
- A `class` is a namespace containing its members
- The public members provide the class's interface and the private members provide implementation details
- A `struct` is a `class` where members are by default `public`

Extract from: Section 16 of "The C++ Programming Language Book by Bjarne Stroustrup"

Example class definition

```
1  class Image {  // Should be in Image.hpp
2   public:
3    Image(const std::string& file_name);
4    void Draw();
5
6   private:
7    int rows_ = 0;  // New in C+=11
8    int cols_ = 0;  // New in C+=11
9  };
10
11 // Implementation omitted here, should be in Image.cpp
12 int main() {
13   Image image("some_image.pgm");
14   image.Draw();
15   return 0;
16 }
```

Classes in our field

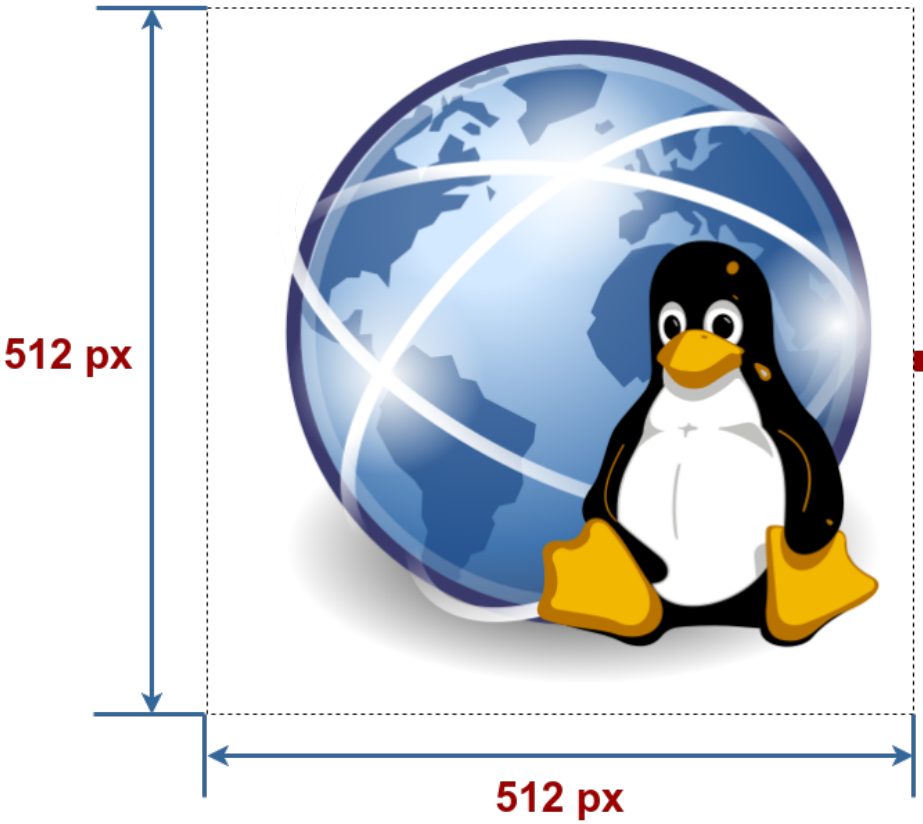
```
// 2D entities
class Image : public Geometry2D;
class RGBDImage : public Geometry2D;

// 3D entities
class Image : public Geometry2D;
class OrientedBoundingBox : public Geometry3D;
class AxisAlignedBoundingBox : public Geometry3D;
class LineSet : public Geometry3D;
class MeshBase : public Geometry3D;
class Octree : public Geometry3D;
class PointCloud : public Geometry3D;
class VoxelGrid : public Geometry3D;

// 3D surfaces
class TetraMesh : public MeshBase;
class TriangleMesh : public MeshBase;
```

Image class

Real Word Entity



Abstraction

```
class Image {
  int rows;
  int cols;
  int num_channels;
  vector<bytes> data;

  // more attributes
}

int main() {
  Image linux_pic("linux.png);

  linux_pic.DrawToScreen();
  linux_pic.ToGrayScale();

  return 0;
}
```

One possible realization

Open3D::Geometry::Image

```
class Image : public Geometry2D {
 public:
  /// Width of the image.
  int width_ = 0;
  /// Height of the image.
  int height_ = 0;
  /// Number of chanels in the image.
  int num_of_channels_ = 0;
  /// Image storage buffer.
  std::vector<uint8_t> data_;
};
```

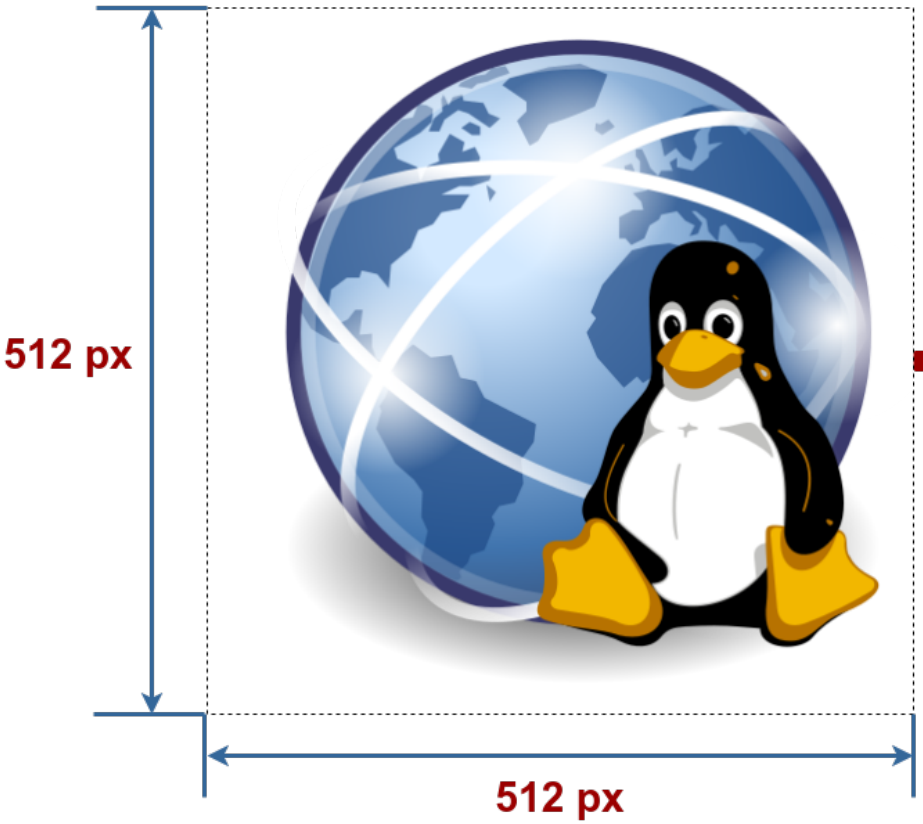
One possible realization

Open3D::Geometry::Image

```
1  class Image : public Geometry2D {
2   public:
3    void Clear() const override;
4    bool IsEmpty() const override;
5
6    Image FlipHorizontal() const;
7    Image FlipVertical() const;
8    Image Filter(Image::FilterType type) const;
9
10  protected:
11   void AllocateDataBuffer() {
12     data_.resize(width_ *
13                  height_ *
14                  num_of_channels_);
15   }
16 }
```

Goal achived?

Real Word Entity



Abstraction

```
class Image {
  int rows;
  int cols;
  int num_channels;
  vector<bytes> data;

  // more attributes
}

int main() {
  Image linux_pic("linux.png);

  linux_pic.DrawToScreen();
  linux_pic.ToGrayScale();

  return 0;
}
```

Goal achived?

Open3D::Geometry::Image

```cpp
#include <Open3D/Geometry/Image.h>

using namespace Open3D::Geometry;
int main() {
  Image linux_pic(".data/linux.png");

  auto flipped_linux = linux_pic.FlipHorizontal();

  auto sobel_filter = Image::FilterType::Sobel3Dx;
  auto filtered_linux = linux_pic.Filter(sobel_filter);

  if (filtered_linux.IsEmpty()) {
    std::cerr << "Couldn't Filter Image!\n";
  }
}
```

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
- **Classes**
  https://en.cppreference.com/w/cpp/classes