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

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

    std::swap(a, b);

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

**Output:**

```
3 5
5 3
```
```c++
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:**

```
1 12
2 3.14
3 12
```
```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
:(
```
```cpp
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
```
```cpp
#include <chrono>

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

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:

```c++
// 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

```c++
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
- [https://google.github.io/styleguide/cppguide.html#Exceptions](https://google.github.io/styleguide/cppguide.html#Exceptions)
Reading and writing to files

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

```cpp
#include <fstream>
using std::string;
using Mode = std::ios_base::openmode;

// ifstream: stream for input from file
std::ifstream f_in(string& file_name, Mode mode);

// ofstream: stream for output to file
std::ofstream f_out(string& file_name, Mode mode);

// stream for input and output to file
std::fstream f_in_out(string& file_name, Mode mode);
```
There are many modes under which a file can be opened

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ios_base::app</td>
<td>append output</td>
</tr>
<tr>
<td>ios_base::ate</td>
<td>seek to EOF when opened</td>
</tr>
<tr>
<td>ios_base::binary</td>
<td>open file in binary mode</td>
</tr>
<tr>
<td>ios_base::in</td>
<td>open file for reading</td>
</tr>
<tr>
<td>ios_base::out</td>
<td>open file for writing</td>
</tr>
<tr>
<td>ios_base::trunc</td>
<td>overwrite the existing file</td>
</tr>
</tbody>
</table>
Regular columns

Use it when:
- The file contains organized data
- Every line has to have all columns

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
#include <fstream> // For the file streams.
#include <iostream>
#include <string>
using namespace std; // Saving space.

int main() {
    int i;
    double a, b;
    string s;
    // Create an input file stream.
    ifstream in("test_cols.txt", ios_base::in);
    // Read data, until it is there.
    while (in >> i >> a >> s >> b) {
        cout << i << " , " << a << " , " << s << " , " << b << endl;
    } return (0);
}
Reading files one line at a time

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

```plaintext
HEADER
a = 4.5
filename = /home/ivizzo/.bashrc
2.34
1 2.23
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

```cpp
#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;
}
```
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**

```c
1 file.write(reinterpret_cast<char*>(a), sizeof(a));
```
Writing to binary files

```cpp
#include <fstream> // for the file streams
#include <vector>
using namespace std;

int main() {
    string file_name = "image.dat";
    ofstream file(file_name, ios_base::out | ios_base::binary);
    int rows = 2;
    int cols = 3;
    vector<float> vec(rows * cols);
    file.write(reinterpret_cast<char*>(&rows), sizeof(rows));
    file.write(reinterpret_cast<char*>(&cols), sizeof(cols));
    file.write(reinterpret_cast<char*>(&vec.front()),
               vec.size() * sizeof(float));
    return 0;
}
```
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**

```cpp
file.read(reinterpret_cast<char*>(a), sizeof(a));
```
# include <fstream>
#include <iostream>
#include <vector>

using namespace std;

int main() {
    string file_name = "image.dat";
    int r = 0, c = 0;
    ifstream in(file_name,
                ios_base::in | ios_base::binary);
    if (!in) { return EXIT_FAILURE; }
    in.read(reinterpret_cast<char*>(&r), sizeof(r));
    in.read(reinterpret_cast<char*>(&c), sizeof(c));
    cout << "Dim: " << r << " x " << c << endl;
    vector<float> data(r * c, 0);
    in.read(reinterpret_cast<char*>(&data.front()),
            data.size() * sizeof(data.front()));
    for (float d : data) { cout << d << endl; }
    return 0;
}
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.
```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() << '
';
    }
    fs::remove_all("sandbox");
}
```

**Output:**
1 "sandbox/a"
2 "sandbox/file1.txt"
3 "sandbox/file2.txt"
```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"
"
"."
".."
```
```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"
```
```cpp
#include <filesystem>
namespace fs = std::filesystem;

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

Output:
```
".txt"
"."
"
".png"
```
```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"
```
```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"
```
void demo_exists(const fs::path& p) {
    cout << p;
    if (fs::exists(p)) cout << " exists\n";
    else cout << " does not exist\n";
}

int main() {
    fs::create_directory("sandbox");
ofstream("sandbox/file"); // create regular file
demo_exists("sandbox/file");
demo_exists("sandbox/cacho");
fs::remove_all("sandbox");
}

Output:
"sandbox/file" exists
"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

```c
void blink_led_bad(int time_to_blink) {
    // do something with time_to_blink
}
```

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

good – the unit is explicit

```c
void blink_led_good(milliseconds time_to_blink) {
    // do something with time_to_blink
}
```

Example taken from: https://youtu.be/fX2W3mNjJIo
Type safety

**good – the unit is explicit**

```c
void blink_led_good(miliseconds time_to_blink) {
    // do something with time_to_blink
}
```

**Usage**

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

Example taken from: https://youtu.be/fX2W3nNjJIo
Want more flexibility?

```cpp
template <class rep, class period>
void blink_led(duration<rep, period> blink_time) {
    // millisecond is the smallest relevant unit
    auto x_ms = duration_cast<miliseconds>(blink_time);
    // do something else with x_ms
}

void use() {
    blink_led(2s); // Works fine
    blink_led(150ms); // Also, works fine
    blink_led(150); // ERROR, which unit?
}
```

Example taken from: https://youtu.be/fX2W3nNjJIo
Type safety in our field

BAD Example: ROS 1

```cpp
// ...
// %Tag(LOOP_RATE)%
ros::Rate loop_rate(10);
// %EndTag(LOOP_RATE)%
// ...
```

`loop_rate` in which units? Hz, ms ???
Type safety in our field

GOOD Example: ROS 2

```
// ...
//
timer_ = create_wall_timer(100ms, timer_callback);
//
// ...
```

- 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
“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”
“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.”
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

```cpp
class Image { // Should be in Image.hpp
    public:
        Image(const std::string& file_name);
        void Draw();

    private:
        int rows_ = 0; // New in C++11
        int cols_ = 0; // New in C++11
};

// Implementation omitted here, should be in Image.cpp
int main() {
    Image image("some_image.pgm");
    image.Draw();
    return 0;
}
```
Classes in our field

1 // 2D entities
2 class Image : public Geometry2D;
3 class RGBDImage : public Geometry2D;
4
5 // 3D entities
6 class Image : public Geometry2D;
7 class OrientedBoundingBox : public Geometry3D;
8 classAxisAlignedBoundingBox : public Geometry3D;
9 class LineSet : public Geometry3D;
10 class MeshBase : public Geometry3D;
11 class Octree : public Geometry3D;
12 class PointCloud : public Geometry3D;
13 class VoxelGrid : public Geometry3D;
14
15 // 3D surfaces
16 class TetraMesh : public MeshBase;
17 class TriangleMesh : public MeshBase;
Image class

```cpp
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 channels in the image.
    int num_of_channels_ = 0;

    /// Image storage buffer.
    std::vector<uint8_t> data_;
};
One possible realization

**Open3D::Geometry::Image**

class Image : public Geometry2D {
public:
    void Clear() const override;
    bool IsEmpty() const override;

    Image FlipHorizontal() const;
    Image FlipVertical() const;
    Image Filter(Image::FilterType type) const;

protected:
    void AllocateDataBuffer() {
        data_.resize(width_ * height_ * num_of_channels_);
    }
};
Goal achieved?

Real Word Entity

Abstraction

```cpp
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 achieved?

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