

Modern C++ for Computer Vision and Image Processing

Lecture 0: The basics

Ignacio Vizzo and Cyrill Stachniss

Course Organization

- **Lectures:** Wednesday 16:00 (CEST)
 - Held at Youtube live-stream on the course channel.
 - Questions via Youtube channel during the lecture.
- **Tutorials:** Friday 15:00 (CEST)
 - Also offline Tutorials.
 - Also "on-demand" Tutorials.
 - Not all the Tutorials are provided by me.
- **Discord:** Fastest channel to discuss.

Course structure

The course is split in **two parts**:

1. Learning the basics

- **Lectures** : Consists of 10 lectures.
- **Homeworks**: Consists of 9 **hands-on** homeworks.

2. Working on a project

- Plan and code **inverse image search**
- Groups of 2 people

Workload

- **180 h** per semester (Workload)
- **60 h** per semester (Lectures)
- **16 weeks** per semester

Doing some math:

$$\left(\frac{180 - 60}{16} \right) \approx 8 \left[\frac{h}{\text{week}} \right]$$

What you will learn in course

- How to work in Linux
- How to write software with modern C++
- Core software development techniques
- How to work with images using [OpenCV](#)
- How to implement **inverse image search**

Check out **Google Image Search** for example: <https://images.google.com/>

How is the course structured?

- **Part I:** C++ basics tools.
- **Part II:** The C++ core language.
- **Part III:** Modern C++.
- **Part IV:** Final project.

Week	Date	Lecture	Homework	Recommended Deadline	Official Deadline
Part I: C++ tools					
-	8-Apr	[[No Lectures]]	-	-	-
0	15-Apr	Course Introduction, Organization, Hello world	-	-	-
1	22-Apr	C++ Tools	Homework 1	3-May	10-May
Part II: The C++ core language					
2	29-Apr	C++ Basic syntax	Homework 2	10-May	17-May
3	6-May	C++ Functions	Homework 3	17-May	24-May
4	13-May	C++ Containers	Homework 4	24-May	31-May
5	20-May	C++ STL Library	Homework 5	31-May	7-Jun
Part III: Modern C++					
6	27-May	Classes	Homework 6	7-Jun	14-Jun
7	3-Jun	OOP	Homework 7	14-Jun	21-Jun
8	10-Jun	Memory Management	Homework 8	21-Jun	28-Jun
9	17-Jun	Generics Programming	Homework 9	28-Jun	5-Jul
Part IV: Final Project "Place recognition using Bag of Visual Words in C++"					
10	24-Jun	Bag of Visual Words			
11	1-Jul	[[No Lectures]]			
12	8-Jul		Final Project	Final Examination Date	
13	15-Jul				

Course Content

Tools

- GNU/Linux [\[Tutorial\]](#)
 - Filesystem
 - Terminal
 - standard input/output
- Text Editor
 - Configuring
 - Terminal
 - Compile
 - Debug
- Build systems
 - headers/sources
 - Libraries
 - Compilation flags
 - CMake
 - 3rd party libraries
- Git [\[Tutorial\]](#)
- Homework submissions
- Gdb [\[Tutorial\]](#)
- Web-based tools
 - Quick Bench
 - Compiler Explorer
 - Cpp insights
 - Cppreference.com
- Clang-tools [\[Tutorial\]](#)
 - Clang-format
 - Clang-tidy
 - Clangd
 - Cppcheck
- Google test [\[tutorial\]](#)
- OpenCV [\[tutorial\]](#)

Core C++

- C++ basic syntax
- The “main” function
- #include statements
- Variables
- Control structures (if, for, while)
- I/O streams
- Input parameters
- Built-in types
- Operators
- Scopes
- Functions
- C++ strings
- Pass by value / Pass by reference
- Namespaces
- Containers
- std::tuple
- Iterators
- try/catch
- enum classes
- STL library
- STL Algorithms
- Function overloading
- Operator overloading
- String streams
- filesystem

Modern C++

- Classes introduction
- Const correctness
- typedef/using
- static variables /methods
- Move Semantics
- Special Functions
- Singleton Pattern
- Inheritance
- Function Overriding
- Abstract classes
- Interfaces
- Strategy Pattern
- Polymorphism
- Typecasting
- Memory management
- Stack vs Heap
- Pointers
- new/delete
- this pointer
- Memory issues
- RAII
- Smart pointers
- Generic programming
- Template functions
- Template classes
- Static code generation
- lambdas

Course Philosophy

Talk is cheap.
Show me the code.

Linus Torvalds

quotefancy

What you will do in this course

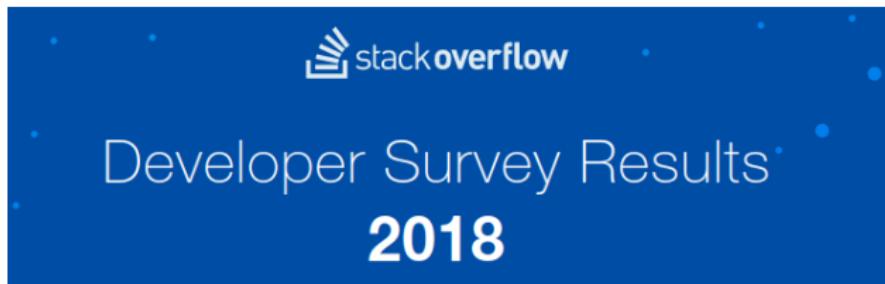


Please stop me!



Why?

Why C++? Why Linux? Why?



- Over 50 000 developers surveyed
- Nearly half of them use Linux
- C++ is the most used systems language (4.5 million users in 2015)
- C++ 11 is a modern language
- All companies want C++ in our field

Stack Overflow survey: <https://insights.stackoverflow.com/survey/2018/>

CLion survey: <https://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/>

Why C++

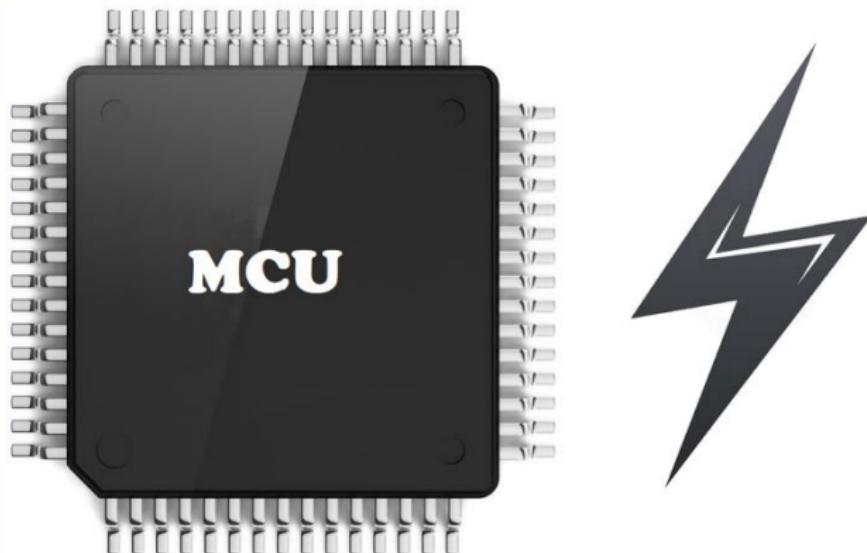


Image taken from <https://circuitdigest.com/>

Companies that use C++



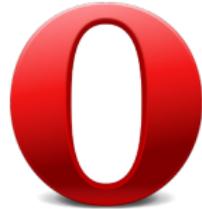
Microsoft



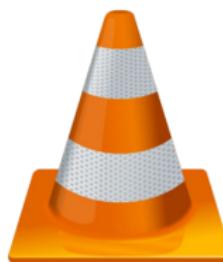
The following slides are adapted from Avery Wang

More info at <http://www.stroustrup.com/applications.html>

Browsers written in C++



Software written in C++



Games written in C++



C++ History: assembly

Benefits:

- Unbelievably simple instructions
- **Extremely** fast (when well-written)
- Complete control over your program

Why don't we always use assembly?

C++ History: assembly

```
1 main:                                # @main
2     push    rax
3     mov     edi, offset std::cout
4     mov     esi, offset .L.str
5     mov     edx, 13
6     call    std::basic_ostream<char, std::
char_traits<char> >& std::__ostream_insert<char, std::
::char_traits<char> >(std::basic_ostream<char, std::
char_traits<char> >&, char const*, long)
7         xor    eax, eax
8         pop    rcx
9         ret
10 _GLOBAL__sub_I_example.cpp:          #
 @_GLOBAL__sub_I_example.cpp
11     push    rax
12     mov     edi, offset std::__ioinit
13     call    std::ios_base::Init::Init() [complete
object constructor]
14     mov     edi, offset std::ios_base::Init::~Init
() [complete object destructor]
15     mov     esi, offset std::__ioinit
16     mov     edx, offset __dso_handle
17     pop    rax
18     jmp    __cxa_atexit           # TAILCALL
19 .L.str:
20     .asciz  "Hello, world\n"
```

C++ History: assembly

Drawbacks:

- A lot of code to do simple tasks
- Hard to understand
- Extremely unportable

C++ History: Invention of C

Problem:

- Computers only understand assembly language.

Idea:

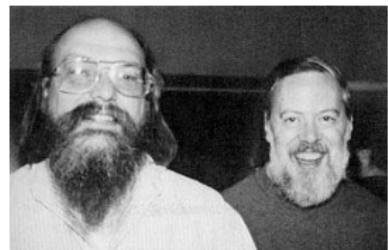
- Source code can be written in a more intuitive language
- An additional program can convert it into assembly [compiler]

C++ History: Invention of C

T&R created **C** in 1972, to much praise.

C made it easy to write code that was

- Fast
- Simple
- Cross-platform



Ken Thompson and Dennis Ritchie, creators of the C language.

C++ History: Invention of C

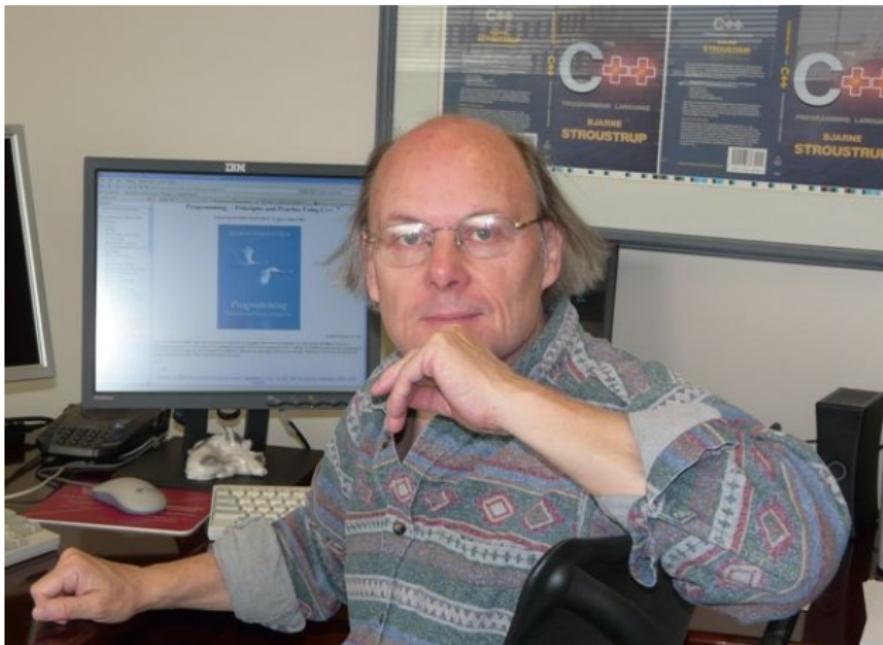
C was popular since it was simple.

This was also its weakness:

- No `objects` or `classes`.
- Difficult to write code that worked `generically`.
- Tedious when writing `large` programs.

C++ History: Welcome to C++

In 1983, the first vestiges of C++ were created by Bjarne Stroustrup.



C++ History: Welcome to C++

He wanted a language that was:

- Fast
- Simple to Use
- Cross-platform
- Had high level features

Evolution of C++

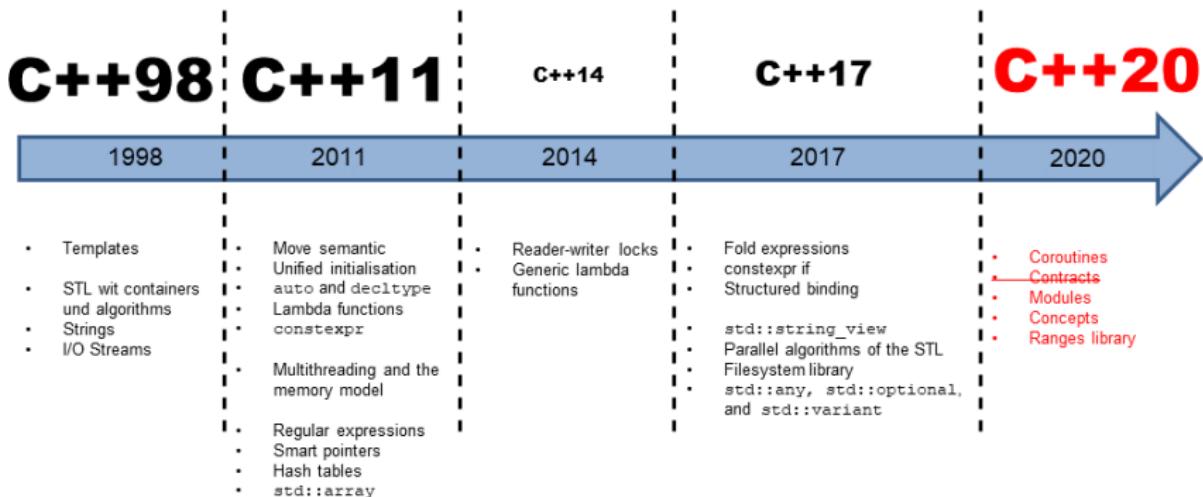
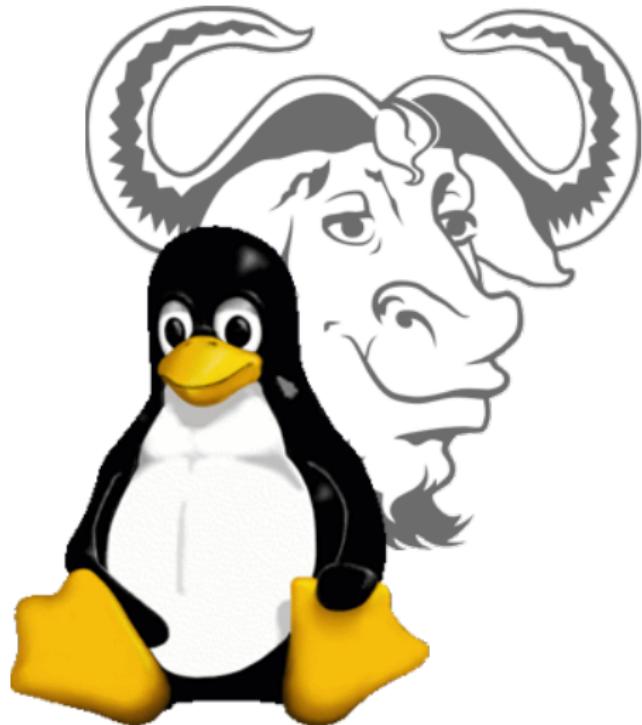


Image taken from <https://www.modernesccpp.com/>

Design Philosophy of C++

- Multi-paradigm
- Express ideas and intent directly in code.
- Safety
- Efficiency
- Abstraction



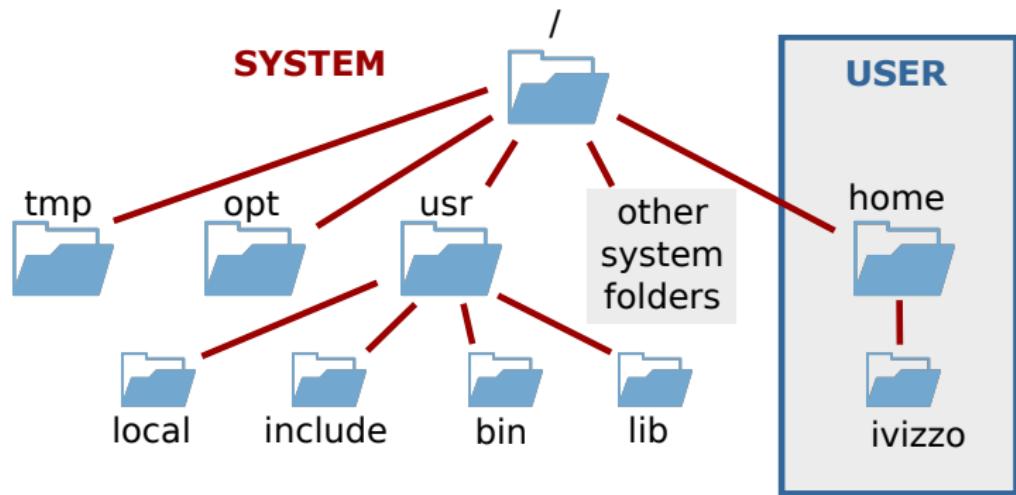
Icon taken from Wikipedia

What is GNU/Linux?

- Linux is a free **Unix-like OS**
- Linux kernel implemented by Linus Torvalds
- **Extremely popular:** Android, ChromeOS, servers, supercomputers, etc.
- Many **Linux distributions** available
- Use any distribution if you have preference
- Examples will be given in **Ubuntu**

ubuntu 

Linux directory tree



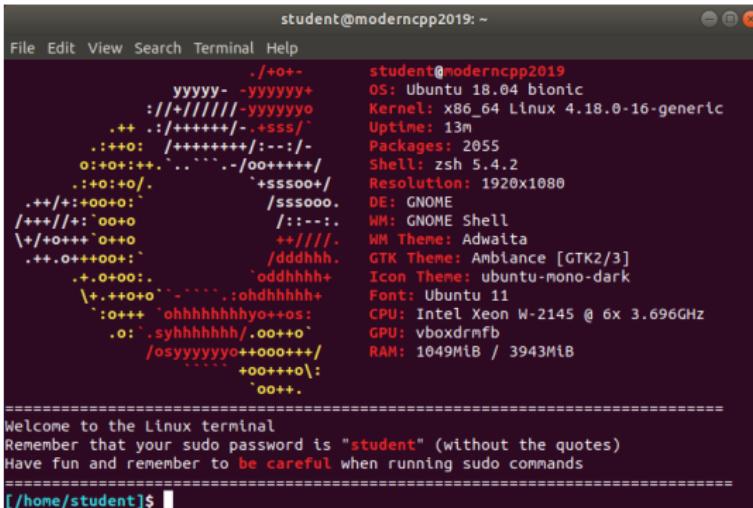
- Tree organization starting with root: /
- There are no volume letters, e.g. C:, D:
- User can only access his/her own folder

Understanding files and folders

- Folders end with `/` e.g. `/path/folder/`
- Everything else is files, e.g. `/path/file`
- Absolute paths start with `/`
while all other paths are relative:
 - `/home/ivizzo/folder/` — **absolute** path to a folder
 - `/home/ivizzo/file.cpp` — **absolute** path to a file
 - `folder/file` — **relative** path to a file
- Paths are case sensitive:
`filename` is different from `FileName`
- Extension is part of a name:
`filename.cpp` is different from `filename.png`

Linux terminal

- Press **Ctrl** + **Alt** + **T** to open terminal



student@moderncpp2019: ~

```
File Edit View Search Terminal Help
student@moderncpp2019
OS: Ubuntu 18.04 bionic
Kernel: x86_64 Linux 4.18.0-16-generic
Uptime: 13m
Packages: 2055
Shell: zsh 5.4.2
Resolution: 1920x1080
DE: GNOME
WM: GNOME Shell
WM Theme: Adwaina
GTK Theme: Ambiance [GTK2/3]
Icon Theme: ubuntu-mono-dark
Font: Ubuntu 11
CPU: Intel Xeon W-2145 @ 6x 3.696GHz
GPU: vboxdrvfb
RAM: 1049MiB / 3943MiB
`oo++o:.

=====
Welcome to the Linux terminal
Remember that your sudo password is "student" (without the quotes)
Have fun and remember to be careful when running sudo commands
=====
[/home/student]$
```

- Most tasks can be done faster from the terminal than from the GUI

Navigating tree from terminal

- Terminal is always in some folder
- `pwd`: **p**rint **w**orking **d**irectory
- `cd <dir>`: **c**hange **d**irectory to `<dir>`
- `ls <dir>`: **l**ist contents of a directory
- Special folders:
 - `/` — root folder
 - `~` — home folder
 - `.` — current folder
 - `..` — parent folder

Structure of Linux commands

Typical structure

`${PATH}/command [options] [parameters]`

- `${PATH}/command`: absolute or relative path to the program binary
- `[options]`: program-specific options
e.g. `-h`, or `--help`
- `[parameters]`: program-specific parameters
e.g. input files, etc.

Use help with Linux programs

- **man** <command> — **man**ual
exhaustive manual on program usage
- **command -h/--help**
usually shorter help message

```
1 [/home/student]$ cat --help
2 Usage: cat [OPTION]... [FILE]...
3 Concatenate FILE(s) to standard output.
4 -A, --show-all           equivalent to -vET
5 -b, --number-nonblank    number nonempty output lines
6
7 Examples:
8   cat f -   Output fs contents, then standard input.
9   cat      Copy standard input to standard output.
```

Using command completion

Pressing  while typing:

- completes name of a file, folder or program
- “beeps” if current text does not match any file or folder uniquely

Pressing  **twice** shows all potential matches

Example:

```
1 [/home/student]$ cd D [TAB] [TAB]
2 Desktop/    Documents/  Downloads/
```

Files and folders

- **mkdir** [-p] <foldername> — **make directory**
Create a folder <foldername>
(with all parent folders [-p])
- **rm** [-r] <name> — **remove [recursive]**
Remove file or folder <name>
(With folder contents [-r])
- **cp** [-r] <source> <dest> — **copy**
Copy file or folder from <source> to <dest>
- **mv** <source> <dest> — **move**
Move file or folder from <source> to <dest>

Using placeholders

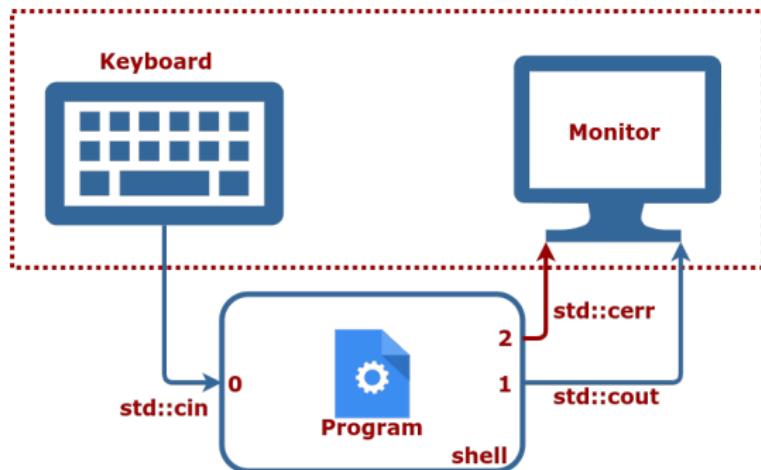
Placeholder	Meaning
*	Any set of characters
?	Any single character
[a-f]	Characters in [abcdef]
[^a-c]	Any character not in [abc]

Can be used with most of terminal commands: `ls`, `rm`, `mv` etc.

```
1 [/home/student/Examples/placeholders]$ ls
2 u01.tex      v01.pdf      v01.tex
3 u02.tex      v02.pdf      v02.tex
4 u03.tex      v03.pdf      v03.tex
5
6 [/home/student/Examples/placeholders]$ ls *.pdf
7 v01.pdf  v02.pdf  v03.pdf
8
9 [/home/student/Examples/placeholders]$ ls u*
10 u01.tex  u02.tex  u03.tex
11
12 [/home/student/Examples/placeholders]$ ls ?01*
13 u01.tex  v01.pdf  v01.tex
14
15 [/home/student/Examples/placeholders]$ ls [uv]01*
16 u01.tex  v01.pdf  v01.tex
17
18 [/home/student/Examples/placeholders]$ ls u0[^12].tex
19 u03.tex
```

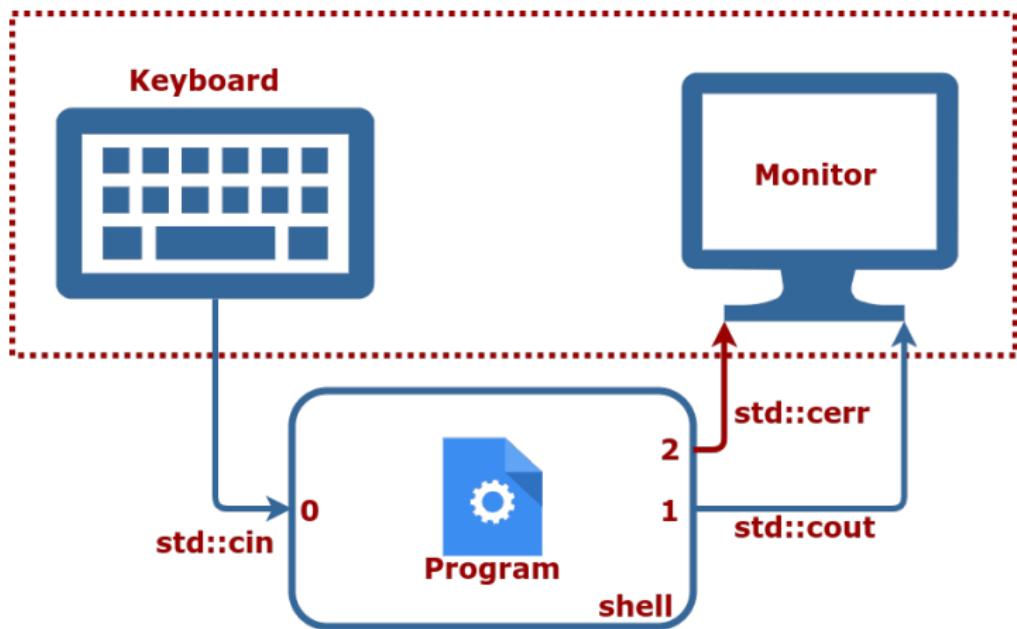
Standard input/output channels

- Single input channel:
 - `stdin`: **Standard input**: channel 0
- Two output channels:
 - `stdout`: **Standard output**: channel 1
 - `stderr`: **Standard error output**: channel 2



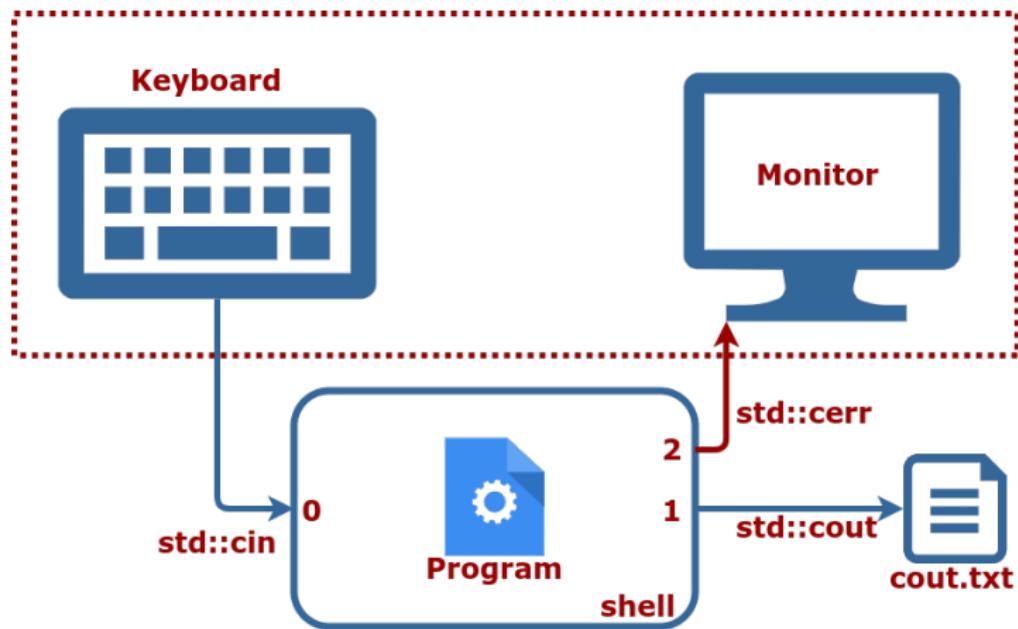
Standard input/output channels

\$ program



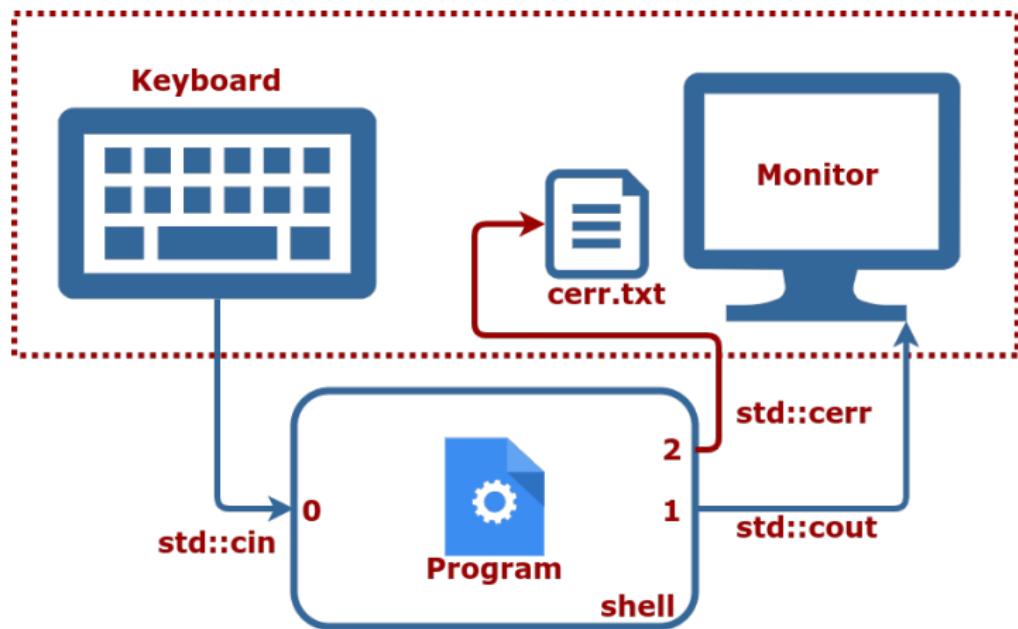
Redirecting `stdout`

```
$ program 1>cout.txt
```



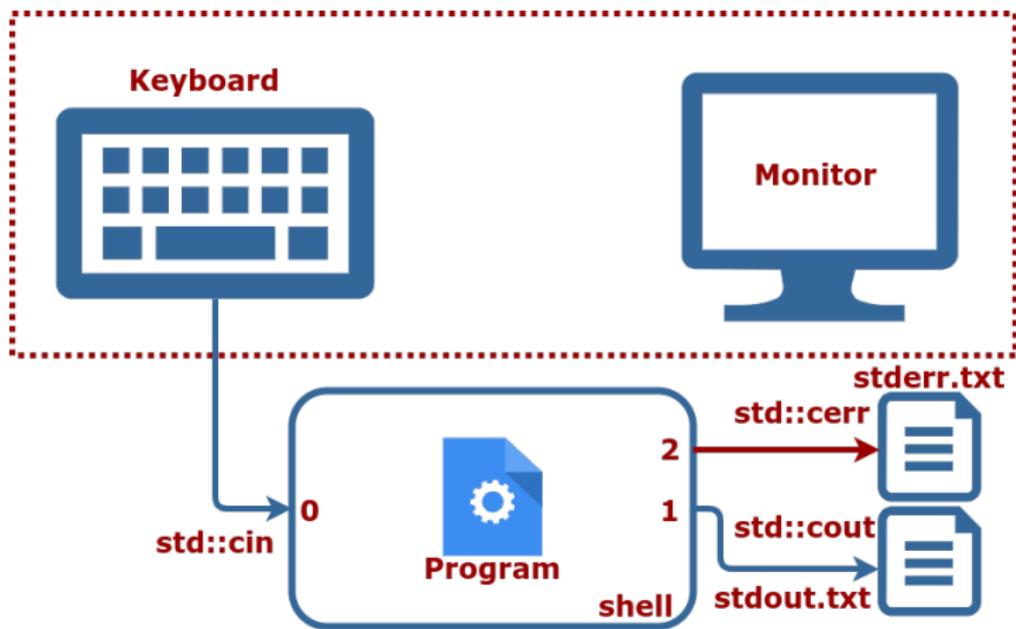
Redirecting stderr

```
$ program 2>cerr.txt
```



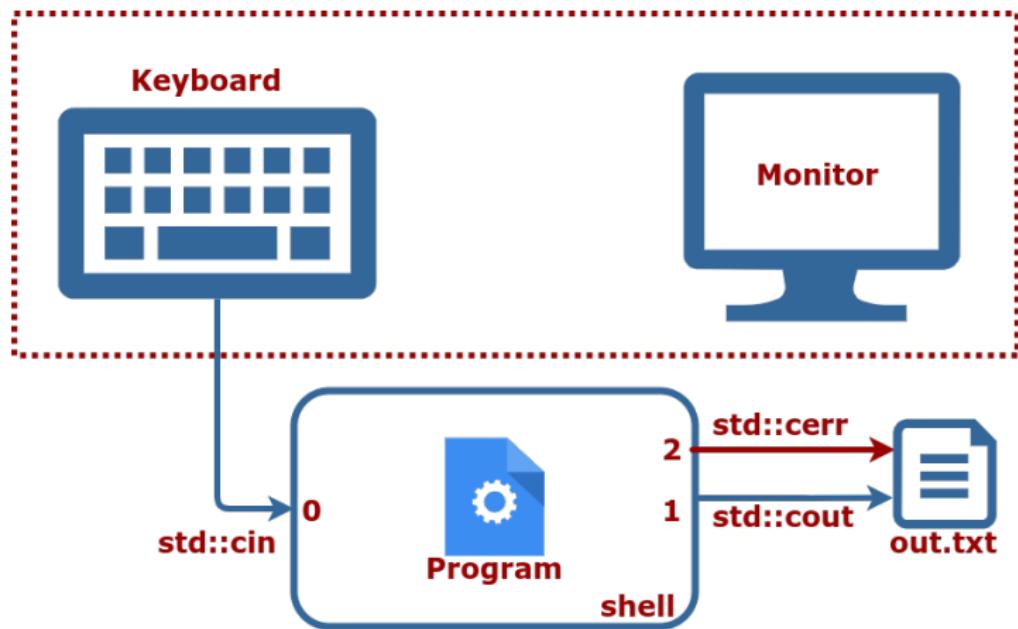
Redirect stdout and stderr

```
$ program 1>stdout.txt 2>stderr.txt
```



Redirect stdout and stderr

programm 1>out.txt 2>&1



Working with files

- **more/less/cat** <filename>
Print the contents of the file
Most of the time using **cat** if enough
- **find** <in-folder> -name <filename>
Search for file <filename> in folder
<in-folder>, allows wildcards
- **locate** <filename>
Search for file <filename> in the entire
system!
just remember to **sudo updatedb** often
- **grep** <what> <where>
Search for a string <what> in a file <where>
- **ag** <what> <where>
Search for a string <what> in a dir <where>

Chaining commands

- `command1; command2; command3`
Calls commands one after another
- `command1 && command2 && command3`
Same as above but fails if any of the commands returns an error code
- `command1 | command2 | command3`
Pipe `stdout` of `command1` to `stdin` of `command2` and `stdout` of `command2` to `stdin` of `command3`
- Piping commonly used with `grep`:
`ls | grep smth` look for `smth` in output of `ls`

Linux Command Line Pipes and Redirection



https://youtu.be/mV_8GbzwZMM

Cancelling commands

- `CTRL + C`
Cancel currently running command
- `kill -9 <pid>`
Kill the process with id `pid`
- `killall <pname>`
Kill all processes with name `pname`
- `htop` (`top`)
 - Shows an overview of running processes
 - Allows to kill processes by pressing `k`

Command history

The shell saves the history of the last executed commands

- : go to the previous command
- : go to the next command
-  +  <query>: search in history
-  + : execute the 10th command
- **history**: show history

Installing software

Most of the software is available in the system repository. To install a program in Ubuntu type this into terminal:

- `sudo apt update` to update information about available packages
- `sudo apt install <program>` to install the program that you want
- Use `apt search <program>` to find all packages that provide `<program>`
- Same for any library, just with `lib` prefix

Bash tutorial

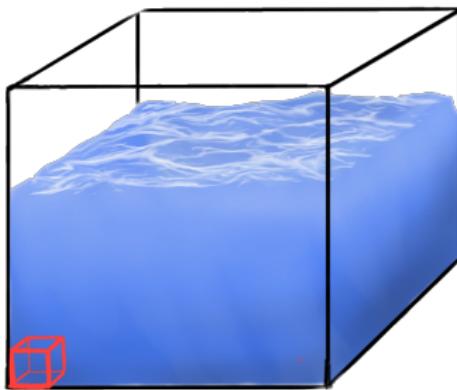


<https://youtu.be/oxuRxtrO2Ag>



Icon taken from Wikipedia

We won't teach you everything about C++



Within C++, there is a much smaller and cleaner language struggling to get out.

-Bjarne Stroustrup

Where to write C++ code

There are two options here:

- Use a C++**IDE**



CLion



Qt Creator



Eclipse

- Use a **modern text editor** [recommended]



Visual Studio Code [my preference]



Sublime Text 3



Atom



VIM [steep learning curve]



Emacs [steep learning curve]

Hello World!

Simple C++ program that prints `Hello World!`

```
1 #include <iostream>
2
3 int main() {
4     // Is this your first C++ program?
5     std::cout << "Hello World!" << std::endl;
6     return 0;
7 }
```

Comments and any whitespace: completely ignored

- A comment is text:
 - On one line that follows `//`
 - Between `/*` and `*/`
- All of these are valid C++:

```
1 int main() {return 0;} // Ignored comment.
```

```
1 int main()
2
3 {
4     return 0;
}
```

```
1 int main() {
2     return /* Ignored comment */ 0;
3 }
```

Good code style is important

Programs are meant to be read by humans and only incidentally for computers to execute.

—Donald Knuth

- Use `clang_format` to format your code
- use `cpplint` to check the style
- Following a style guide will save you time and make the code more readable
- We use **Google Code Style Sheet**
- Naming and style recommendations will be marked by `GOOGLE-STYLE` tag in slides

Everything starts with main

- **Every** C++ program starts with `main`
- `main` is a function that returns an error code
- Error code `0` means `OK`
- Error code can be any number in `[1, 255]`

```
1 int main() {  
2     return 0; // Program finished without errors.  
3 }
```

```
1 int main() {  
2     return 1; // Program finished with error code 1.  
3 }
```

#include directive

Two variants:

- `#include <file>` — system include files
- `#include "file"` — local include files

Copies the content of `file` into the current file

```
1 #include "some_file.hpp"
2 // We can use contents of file "some_file.hpp" now.
3 int main() { return 0; }
```

I/O streams for simple input and output

- Handle `stdin`, `stdout` and `stderr`:
 - `std::cin` — maps to `stdin`
 - `std::cout` — maps to `stdout`
 - `std::cerr` — maps to `stderr`
- `#include <iostream>` to use I/O streams
- Part of C++ standard library

```
1 #include <iostream>
2 int main() {
3     int some_number;
4     std::cout << "please input any number" << std::endl;
5     std::cin >> some_number;
6     std::cout << "number = " << some_number << std::endl;
7     std::cerr << "boring error message" << std::endl;
8     return 0;
9 }
```

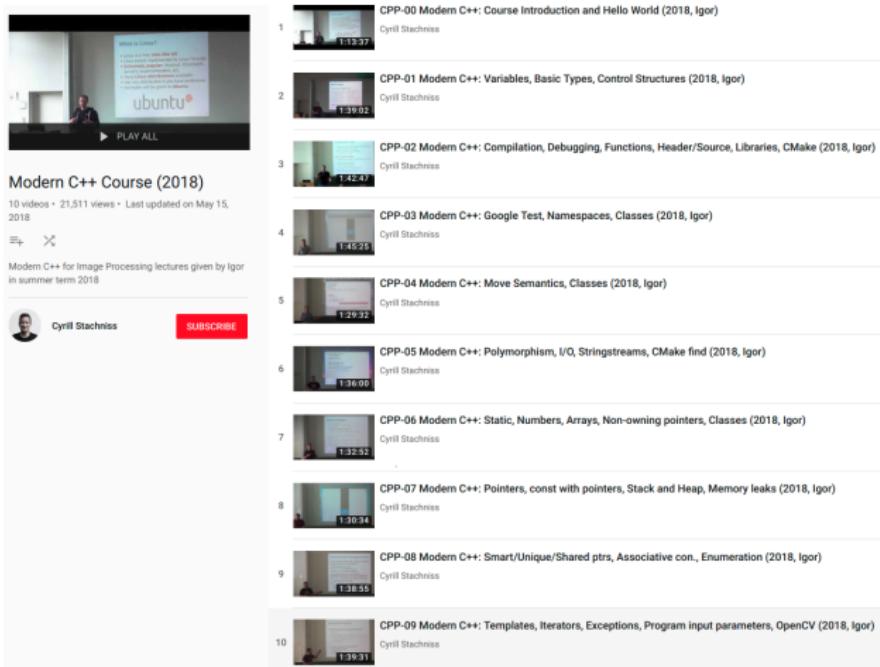
Compile and run Hello World!

- We understand **text**
- Computer understands **machine code**
- **Compilation** is translation from text to machine code
- **Compilers** we can use on Linux:
 - Clang [*] [used in examples]
 - GCC

Compile and run Hello World example:

```
1 c++ -std=c++11 -o hello_world hello_world.cpp
2 ./hello_world
```

Credits to Igor the great



Modern C++ Course (2018)

10 videos • 21,511 views • Last updated on May 15, 2018

Modern C++ for Image Processing lectures given by Igor in summer term 2018

 Cyril Stachniss 

1 CPP-00 Modern C++: Course Introduction and Hello World (2018, Igor)
Cyril Stachniss 1:12:37

2 CPP-01 Modern C++: Variables, Basic Types, Control Structures (2018, Igor)
Cyril Stachniss 1:39:02

3 CPP-02 Modern C++: Compilation, Debugging, Functions, Header/Source, Libraries, CMake (2018, Igor)
Cyril Stachniss 1:42:47

4 CPP-03 Modern C++: Google Test, Namespaces, Classes (2018, Igor)
Cyril Stachniss 1:45:25

5 CPP-04 Modern C++: Move Semantics, Classes (2018, Igor)
Cyril Stachniss 1:29:32

6 CPP-05 Modern C++: Polymorphism, I/O, Streamss, CMake find (2018, Igor)
Cyril Stachniss 1:36:00

7 CPP-06 Modern C++: Static, Numbers, Arrays, Non-owning pointers, Classes (2018, Igor)
Cyril Stachniss 1:32:52

8 CPP-07 Modern C++: Pointers, const with pointers, Stack and Heap, Memory leaks (2018, Igor)
Cyril Stachniss 1:30:34

9 CPP-08 Modern C++: Smart/Unique/Shared ptrs, Associative con., Enumeration (2018, Igor)
Cyril Stachniss 1:38:55

10 CPP-09 Modern C++: Templates, Iterators, Exceptions, Program input parameters, OpenCV (2018, Igor)
Cyril Stachniss 1:39:31

<https://bit.ly/2JmIqGs> [shortened]

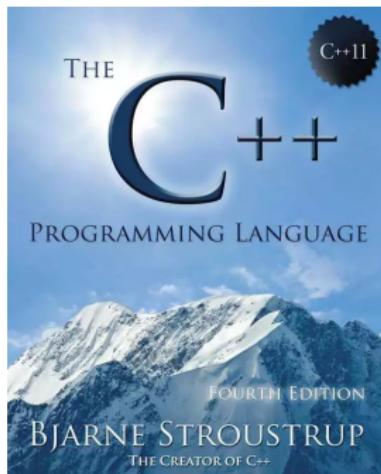
Suggested Video

“You Should Learn to Program” by
Christian Genco at TEDxSMU



<https://youtu.be/xfBWk4nw440>

C++ Programming Language



■ Website:

<http://www.stroustrup.com/4th.html>

Best reference

C++ reference

C++98, C++03, C++11, C++14, C++17, C++20	
Language Basic concepts C++ keywords Preprocessor Expressions Declaration Initialization Functions Statements Classes Templates Exceptions	Compiler support Freestanding implementations
Headers Named requirements Feature test macros (C++20)	Concepts library (C++20) Diagnostics library General utilities library Smart pointers and allocators Date and time Function objects – <code>hash</code> (C++11) String conversions (C++11) Utility functions <code>pair</code> – <code>tuple</code> (C++11) <code>optional</code> (C++17) – <code>any</code> (C++17) <code>variant</code> (C++17) – <code>any</code> (C++17) <code>variant</code> (C++20) – <code>format</code> (C++20)
Language support library Type support – <code>traits</code> (C++11) Program utilities Relational comparators (C++20) <code>numeric_limits</code> – <code>type_info</code> <code>initializer_list</code> (C++11)	Containers library <code>array</code> (C++11) – <code>vector</code> <code>map</code> – <code>unordered_map</code> (C++11) <code>priority_queue</code> – <code>span</code> (C++20) Other containers: <code>sequence</code> – <code>associative</code> <code>unordered_associative</code> – <code>adaptors</code>
Technical specifications Standard library extensions (library fundamentals TS) resource_adaptor – <code>invocation_type</code> Standard library extensions v2 (library fundamentals TS v2) <code>propagate_const</code> – <code>ostream</code> <code>joiner</code> – <code>randint</code> <code>observer_ptr</code> – detection idiom Standard library extensions v3 (library fundamentals TS v3) <code>scope_exit</code> – <code>scope_fail</code> – <code>scope_success</code> – <code>unique_resource</code> Concurrency library extensions (concurrency TS) Concepts (concepts TS) Ranges (ranges TS) Transactional Memory (TM TS)	Iterators library Ranges library (C++20) Algorithms library Numerics library Common math functions Mathematical special functions (C++17) Numeric algorithms Pseudo-random number generation Floating-point environment (C++11) <code>complex</code> – <code>array</code> Input/output library Stream-based I/O Synchronized output (C++20) I/O manipulators Localizations library Regular expressions library (C++11) basic_regex – algorithms Atomic operations library (C++11) atomic – <code>atomic_flag</code> <code>atomic_ref</code> (C++20) Thread support library (C++11) Filesystem library (C++17)
External Links – Non-ANSI/ISO Libraries – Index – std Symbol Index	

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

References

- **C++ Reference:**

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

- **Cpp Core Guidelines:**

<https://github.com/isocpp/CppCoreGuidelines>

- **Google Code Styleguide:**

<https://google.github.io/styleguide/cppguide.html>

- **C++ Tutorial:**

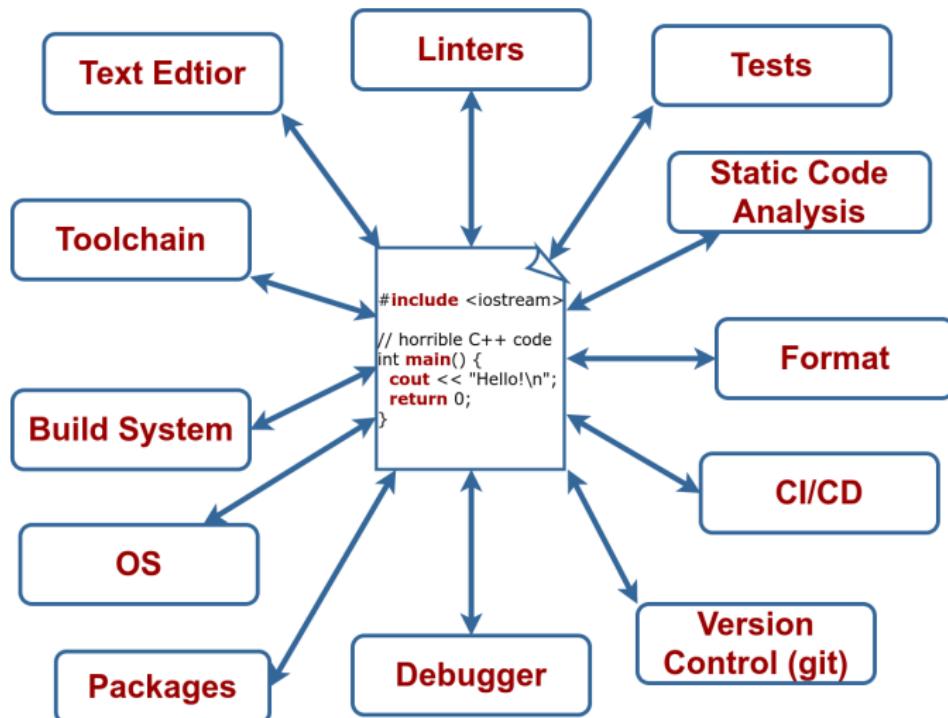
<http://www.cplusplus.com/doc/tutorial/>

Modern C++ for Computer Vision and Image Processing

Lecture 1: Build and Tools

Ignacio Vizzo and Cyrill Stachniss

SW dev ecosystem

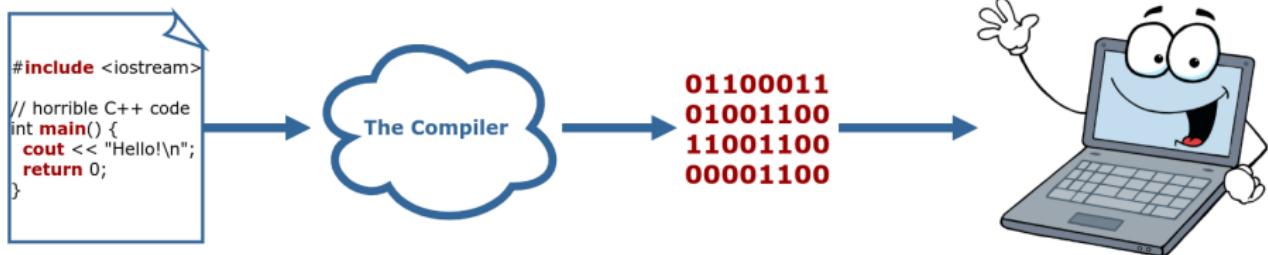


The compilation process

What is a compiler?

- A compiler is basically... a program.
- But not any program.
- Is in charge on transforming your horrible source code into binary code.
- Binary code, `0100010001`, is the language that a computer can understand.

What is a compiler?



Compilation made easy

The easiest compile command possible:

- `clang++ main.cpp`
- This will build a program called `a.out` that it's ready to run.

Will be always this easy?

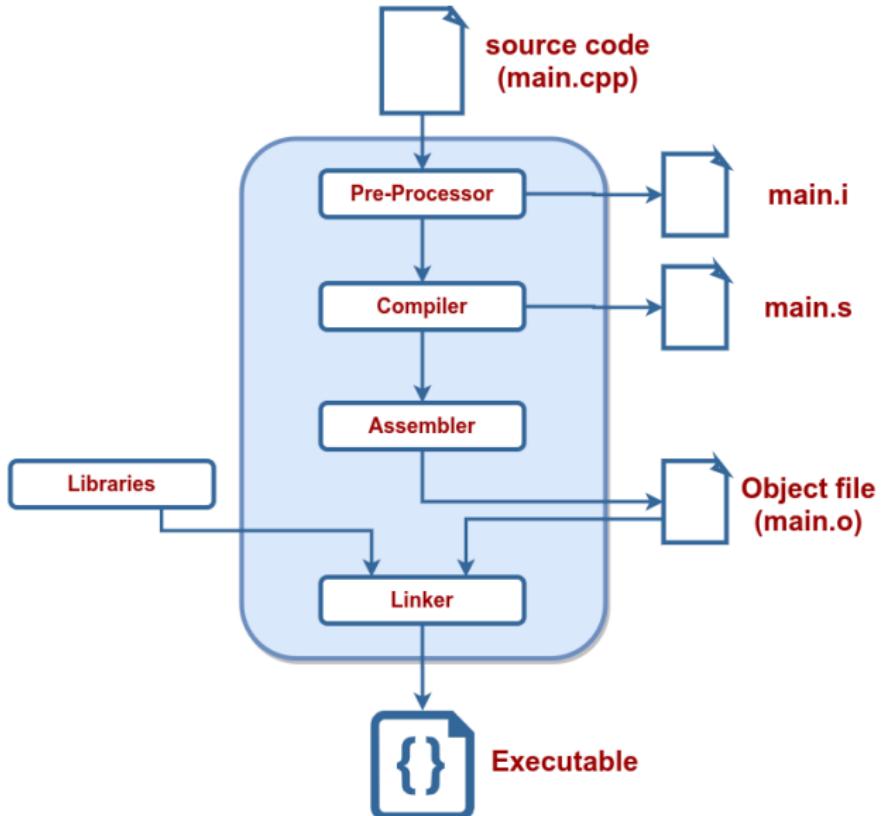
- Of course, not.

The Compiler: Behind the scenes

The compiler performs 4 distinct actions to build your code:

- 1. Pre-process**
- 2. Compile**
- 3. Assembly**
- 4. Link**

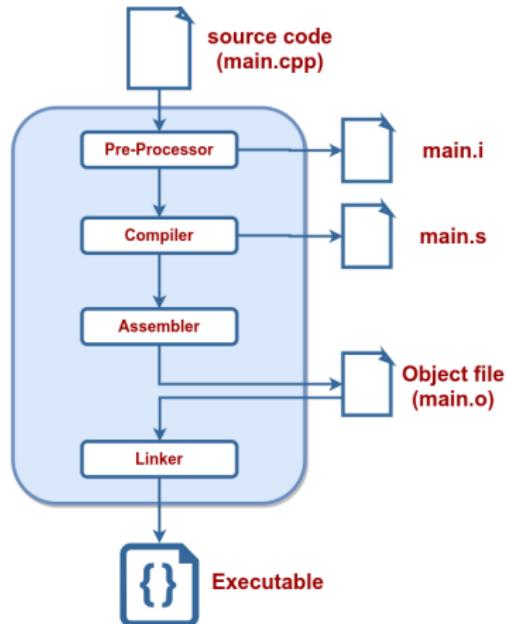
The Compiler: Behind the scenes



Compiling step-by-step

1. Preprocess:

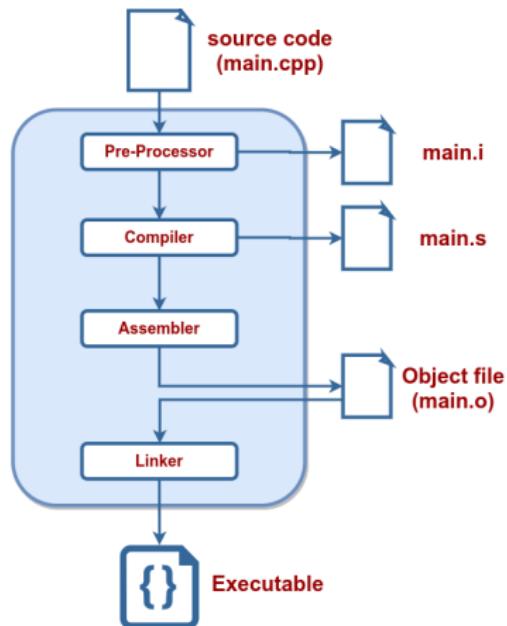
- clang++ -E main.cpp > main.i



Compiling step-by-step

2. Compilation:

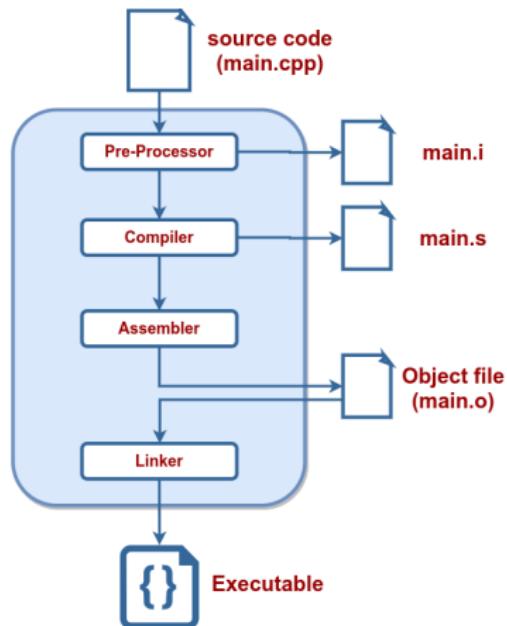
- clang++ -S main.cpp



Compiling step-by-step

3. Assembly:

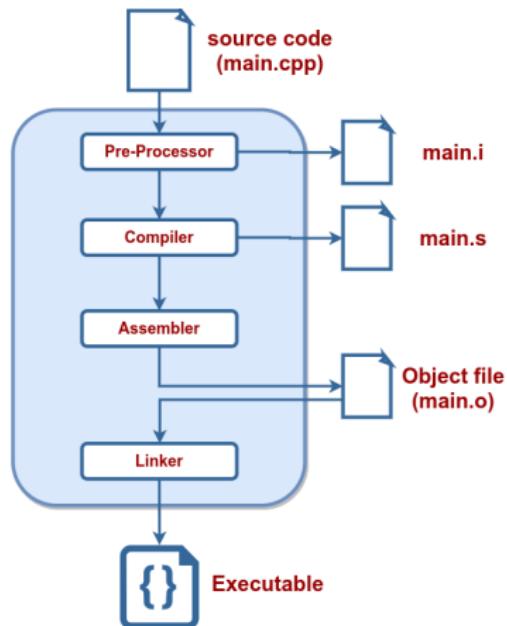
- clang++ -c main.cpp



Compiling step-by-step

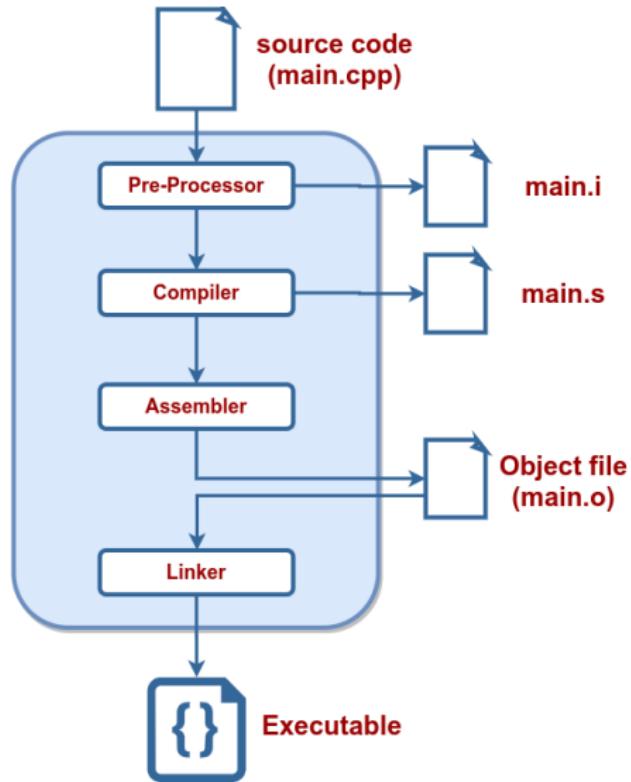
4. Linking:

- clang++ main.o -o main



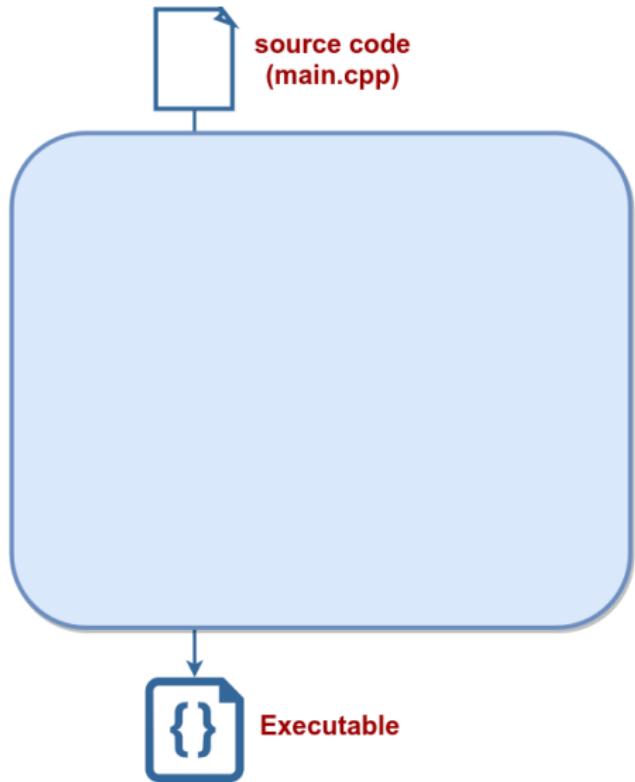
Compiling recap

1. clang++ -E main.cpp
2. clang++ -S main.i
3. clang++ -c main.s
4. clang++ main.o



Compiling recap

1. clang++ main.cpp



Compilation flags

- There is a lot of flags that can be passed while compiling the code
- We have seen some already:
`-std=c++17`, `-o`, etc.

Other useful options:

- Enable all warnings, treat them as errors:
`-Wall`, `-Wextra`, `-Werror`
- Optimization options:
 - `-O0` — no optimizations **[default]**
 - `-O3` or `-Ofast` — full optimizations
- Keep debugging symbols: `-g`

Play with them with Compiler Explorer: <https://godbolt.org/>

Libraries

What is a Library

- Collection of symbols.
- Collection of function implementations.



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/...`

Declaration and definition

- Function declaration can be separated from the implementation details
- Function **declaration** sets up an interface

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

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

```
1 void FuncName(int param) {  
2     // Implementation details.  
3     cout << "This function is called FuncName! ";  
4     cout << "Did you expect anything useful from it?";  
5 }
```

Header / Source Separation

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

```
1 // some_file.hpp
2 Type SomeFunc(... args...);
3
4 // some_file.cpp
5 #include "some_file.hpp"
6 Type SomeFunc(... args...) {} // implementation
7
8 // program.cpp
9 #include "some_file.hpp"
10 int main() {
11     SomeFunc(/* args */);
12     return 0;
13 }
```

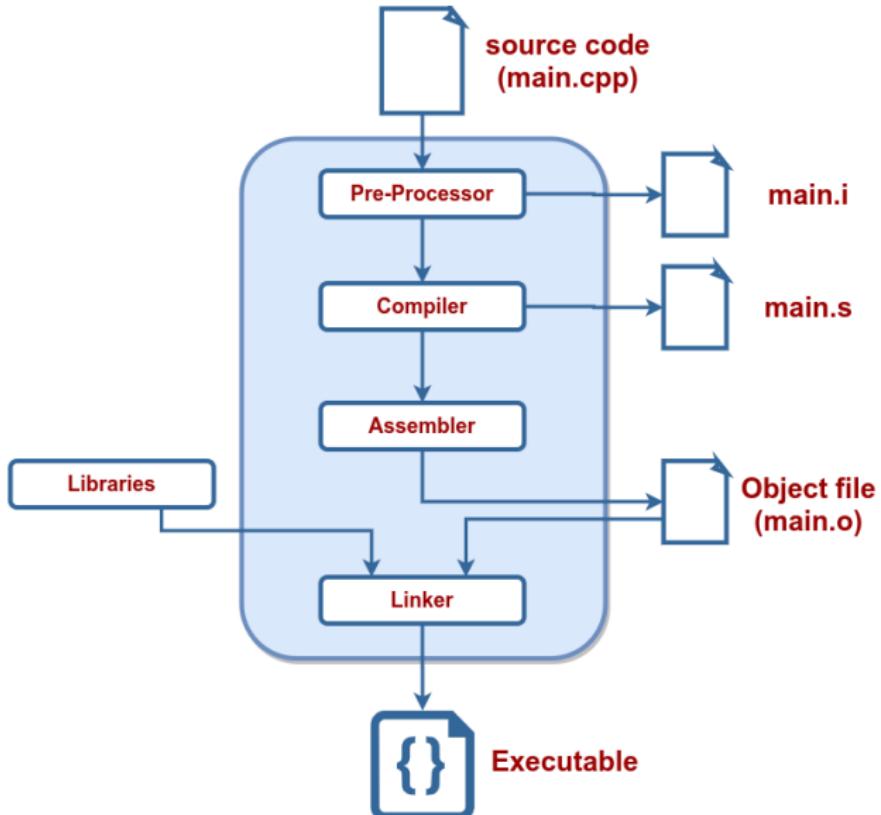
Just build it as before?

```
clang++ -std=c++17 program.cpp -o main
```

Error:

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

What is linking?



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:**
 1. A header file `library_api.h`
 2. The compiled library object `libmylibrary.a`

How to build libraries?

```
1 folder/
2     --- tools.hpp
3     --- tools.cpp
4     --- main.cpp
```

Short: we separate the code into modules

Declaration: tools.hpp

```
1 #pragma once // Ensure file is included only once
2 void MakeItSunny();
3 void MakeItRain();
```

How to build libraries?

Definition: tools.cpp

```
1 #include "tools.hpp"
2 #include <iostream>
3 void MakeItRain() {
4     // important weather manipulation code
5     std::cout << "Here! Now it rains! Happy?\n";
6 }
7 void MakeItSunny() { std::cerr << "Not available\n"; }
```

Calling: main.cpp

```
1 #include "tools.hpp"
2 int main() {
3     MakeItRain();
4     MakeItSunny();
5     return 0;
6 }
```

Use modules and libraries!

Compile modules:

```
c++ -std=c++17 -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++17 main.cpp -L . -ltools -o main
```

Run the code:

```
./main
```

Build Systems

Building by hand is hard

- 4 commands to build a simple hello world example with 2 symbols.
- How does it scales on big projects?
- Impossible to maintain.
- Build systems to the rescue!

What are build systems

- Tools.
- Many of them.
- Automate the build process of projects.
- They began as `shell` scripts
- Then turn into `MakeFiles`.
- And now into MetaBuild Systems like `CMake`.
 - Accept it, `CMake` is not a build system.
 - It's a build system generator
 - You need to use an actual build system like `Make` or `Ninja`.

What I wish I could write

Replace the build commands:

1. `c++ -std=c++17 -c tools.cpp -o tools.o`
2. `ar rcs libtools.a tools.o <other_modules>`
3. `c++ -std=c++17 main.cpp -L . -ltools`

For a script in the form of:

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

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



Build a CMake project

- **Build process** from the user's perspective
 1. `cd <project_folder>`
 2. `mkdir build`
 3. `cd build`
 4. `cmake ..`
 5. `make`
- The build process is completely defined in `CMakeLists.txt`
- And childrens `src/CMakeLists.txt`, etc.

First CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.1) # Mandatory.
2 project(first_project)           # Mandatory.
3 set(CMAKE_CXX_STANDARD 17)        # Use c++17.
4
5 # tell cmake where to look for *.hpp, *.h files
6 include_directories(include)
7
8 # create library "libtools"
9 add_library(tools src/tools.cpp) # creates libtools.a
10
11 # add executable main
12 add_executable(main src/tools_main.cpp) # main.o
13
14 # tell the linker to bind these objects together
15 target_link_libraries(main tools) # ./main
```

CMake is easy to use

- All build **files are in one place**
- The build **script is readable**
- Automatically **detects changes**
- After doing changes:
 1. `cd <project_folder>/build`
 2. `make`

Typical project structure

```
1  |-- project_name/
2  |  |-- CMakeLists.txt
3  |  |-- build/  # All generated build files
4  |  |-- results/ # Executable artifacts
5  |  |  |-- bin/
6  |  |  |  |-- tools_demo
7  |  |  |-- lib/
8  |  |  |  |-- libtools.a
9  |  |-- include/ # API of the project
10 |  |  |-- project_name
11 |  |  |  |-- library_api.hpp
12 |  |-- src/
13 |  |  |-- CMakeLists.txt
14 |  |  |-- project_name
15 |  |  |  |-- CMakeLists.txt
16 |  |  |  |-- tools.hpp
17 |  |  |  |-- tools.cpp
18 |  |  |  |-- tools_demo.cpp
19 |  |-- tests/  # Tests for your code
20 |  |  |-- test_tools.cpp
21 |  |  |-- CMakeLists.txt
22 |  |-- README.md  # How to use your code
```

Compilation options in CMake

```
1 set(CMAKE_CXX_STANDARD 17)
2
3 # Set build type if not set.
4 if(NOT CMAKE_BUILD_TYPE)
5   set(CMAKE_BUILD_TYPE Debug)
6 endif()
7 # Set additional flags.
8 set(CMAKE_CXX_FLAGS "-Wall -Wextra")
9 set(CMAKE_CXX_FLAGS_DEBUG "-g -O0")
```

- `-Wall -Wextra`: show all warnings
- `-g`: keep debug information in binary
- `-O<num>`: optimization level in `{0, 1, 2, 3}`
 - 0: no optimization
 - 3: full optimization

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`

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`

Everything is broken, what should I do?

- 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 -rf *` [make sure you are in build folder]
- Short way (If you are in `project/`):
 - `rm -rf build/`

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:

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

CMake find_path and find_library

- We can use an external library
- Need headers and binary library files
- There is an easy way to find them
- **Headers:**

```
1 find_path(SOME_PKG_INCLUDE_DIR include/some_file.hpp
2           <path1> <path2> ...)
3 include_directories(${SOME_PKG_INCLUDE_DIR})
```

- **Libraries:**

```
1 find_library(SOME_LIB
2               NAMES <some_lib>
3               PATHS <path1> <path2> ...)
4 target_link_libraries(target ${SOME_LIB})
```

`find_package`

- `find_package` calls multiple `find_path` and `find_library` functions
- To use `find_package(<pkg>)` CMake must have a file `Find<pkg>.cmake` in `CMAKE_MODULE_PATH` folders
- `Find<pkg>.cmake` defines which libraries and headers belong to package `<pkg>`
- Pre-defined for most popular libraries, e.g. OpenCV, libpng, etc.

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.1)
2 project(first_project)
3
4 # CMake will search here for Find<pkg>.cmake files
5 SET(CMAKE_MODULE_PATH
6     ${PROJECT_SOURCE_DIR}/cmake_modules)
7
8 # Search for Findsome_pkg.cmake file and load it
9 find_package(some_pkg)
10
11 # Add the include folders from some_pkg
12 include_directories(${some_pkg_INCLUDE_DIRS})
13
14 # Add the executable "main"
15 add_executable(main small_main.cpp)
16 # Tell the linker to bind these binary objects
17 target_link_libraries(main ${some_pkg_LIBRARIES})
```

cmake_modules/Findsome_pkg.cmake

```
1 # Find the headers that we will need
2 find_path(some_pkg_INCLUDE_DIRS include/some_lib.hpp <
3           FOLDER_WHERE_TO_SEARCH>)
4 message(STATUS "headers: ${some_pkg_INCLUDE_DIRS}")
5
5 # Find the corresponding libraries
6 find_library(some_pkg_LIBRARIES
7               NAMES some_lib_name
8               PATHS <FOLDER_WHERE_TO_SEARCH>)
9 message(STATUS "libs: ${some_pkg_LIBRARIES}")
```

Watch for Homeworks



<https://youtu.be/hwP7WQkmECE>

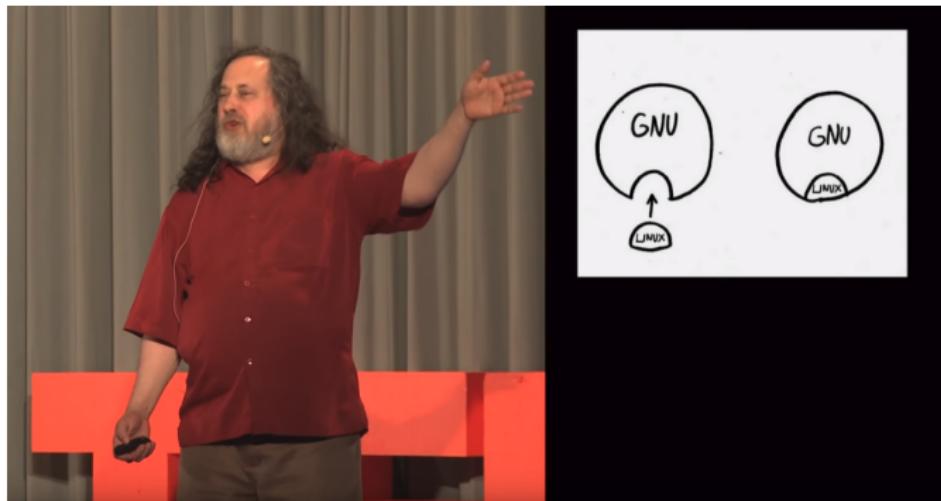
Watch for Homeworks



<https://youtu.be/OZEGnam2M9s>

Suggested Video

“Free software, free society” by Richard Stallman



https://youtu.be/Ag1AKII_2GM

References

- **CMake Documentation**

cmake.org/cmake/help/v3.10/

- **GCC Manual**

gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/

- **Clang Manual**

releases.llvm.org/10.0.0/tools/clang/docs/index.html

Modern C++ for Computer Vision and Image Processing

Lecture 2: Core C++

Ignacio Vizzo and Cyrill Stachniss

C++, the legals



C++ Program

“A C++ program is a sequence of text files (typically header and source files) that contain **declarations**. They undergo **translation** to become an executable program, which is executed when the C++ implementation calls its **main function**.”

C++ Keywords

“Certain words in a C++ program have special meaning, and these are known as **keywords**. Others can be used as **identifiers**. **Comments** are ignored during translation. Certain characters in the program have to be represented with **escape sequences**.”

```
1 const, auto, friend, false, ... //< C++ Keywords
2 // comment type 1
3 /* comment type 2 */
4 /* comment type 3
   BLOCK COMMENT
5 */
6
7 "Hello C++ \n"; //< "\n" is an escape character
```

C++ Entities

“The entities of a C++ program are values, objects, references, functions, enumerators, types, class members, templates, template specializations, namespaces. Preprocessor macros are not C++ entities.”

```
1 3.5f;           // value entity
2 std::string str1; // object entity
3 namespace std;  // namespace entity
4 void MyFunc(); // function entity
5 const int& a = b; // reference entity
6 enum MyEnum {}; // enum entity
7 #define UGLY_MACRO(X) // NOT a C++ entity
```

C++ Declarations

“Declarations may introduce entities, associate them with **names** and define their properties. The **declarations** that define all properties required to use an entity are **definitions**.”

```
1 int foo;           // introduce entity named "foo"  
2  
3 void MyFunc(); // introduce entity named "MyFunc"  
4  
5 // introduce entity named "GreatFunction"  
6 // Also, this is a definition of "GreatFunction",  
7 void GreatFunction() {  
8     // do stuff  
9 }
```

C++ Definitions

“**Definitions** of functions usually include sequences of **statements**, some of which include **expressions**, which specify the computations to be performed by the program.”

```
1 // Function Definition
2 void MyFunction() {
3     int a;           // statement
4     int b;           // statement
5     int c = a + b;  // a + b is an expression
6 }
```

NOTE: Every C++ statement ends with a semicolon “;”

C++ Names

“Names encountered in a program are associated with the declarations that introduced them. Each name is only valid within a part of the program called its **scope**.”

```
1 int my_variable; // "my_variable" is the name
2
3 {
4     float var_fl; // var_f is valid within this scope
5 } //}-this defines end of the scope
6
7 var_fl; // Error, var_fl outside its scope
8
9 int var_fl; // Valid, var_fl not declared
```

C++ Types

“Each object, reference, function, expression in C++ is associated with a **type**, which may be **fundamental**, **compound**, or **user-defined**, **complete** or **incomplete**, etc.”

```
1 float a;           // float is the fundamental type of a
2 bool b;           // bool is fundamental
3
4 MyType c;           // MyType is user defined, incomplete
5 MyType c{};         // MyType is user defined, complete
6
7 std::vector;        // Also, user-defined type
8 std::string;        // Also, user-defined type
```

C++ Variables

“Declared objects and declared references are variables, except for **non-static** data members.”

```
1 int foo;           // variable
2 bool know_stuff;  // also, variable
3
4 MyType my_var;    // variable
5 MyType::var;      // static data member, variable
6 MyType.data_member; // non-static data member
```

C++ Identifiers

“An identifier is an arbitrarily long sequence of digits, underscores, lowercase and uppercase Latin letters, and most Unicode characters. A valid identifier must begin with a **non-digit**. Identifiers are case-sensitive.”

```
1 int s_my_var;    // valid identifier
2 int S_my_var;    // valid but different
3 int SMYVAR;      // also valid
4 int A_6_;        // valid
5 int Ü_ß_vär;    // valid
6 int 6_a;         // NOT valid, illegal
7 int this_identifier_sadly_is_consider_valid_but_long;
```

C++ Keywords

alignas (since C++11)	default(1)	register(2)
alignof (since C++11)	delete(1)	reinterpret_cast
and	do	requires (since C++20)
and_eq	double	return
asm	dynamic_cast	short
atomic_cancel (TM TS)	else	signed
atomic_commit (TM TS)	enum	sizeof(1)
atomic_noexcept (TM TS)	explicit	static
auto(1)	export(1)(3)	static_assert (since C++11)
bitand	extern(1)	static_cast
bitor	false	struct(1)
bool	float	switch
break	for	synchronized (TM TS)
case	friend	template
catch	goto	this
char	if	thread_local (since C++11)
char8_t (since C++20)	inline(1)	throw
char16_t (since C++11)	int	true
char32_t (since C++11)	long	try
class(1)	mutable(1)	typedef
compl	namespace	typeid
concept (since C++20)	new	typename
const	noexcept (since C++11)	union
constexpr (since C++20)	not	unsigned
constexpr (since C++11)	not_eq	using(1)
constinit (since C++20)	nullptr (since C++11)	virtual
const_cast	operator	void
continue	or	volatile
co_await (since C++20)	or_eq	wchar_t
co_return (since C++20)	private	while
co_yield (since C++20)	protected	xor
decltype (since C++11)	public	xor_eq
	reflexpr (reflection TS)	

C++ Expressions

“An expression is a sequence of operators and their operands, that specifies a computation.”

Common operators						
assignment	increment decrement	arithmetic	logical	comparison	member access	other
<code>a = b</code> <code>a += b</code> <code>a -= b</code> <code>a *= b</code> <code>a /= b</code> <code>a %= b</code> <code>a &= b</code> <code>a = b</code> <code>a ^= b</code> <code>a <= b</code> <code>a >= b</code>	<code>++a</code> <code>--a</code> <code>a++</code> <code>a--</code>	<code>+a</code> <code>-a</code> <code>a + b</code> <code>a - b</code> <code>a * b</code> <code>a / b</code> <code>a % b</code> <code>~a</code> <code>a & b</code> <code>a b</code> <code>a ^ b</code> <code>a << b</code> <code>a >> b</code>	<code>!a</code> <code>a && b</code> <code>a b</code>	<code>a == b</code> <code>a != b</code> <code>a < b</code> <code>a > b</code> <code>a <= b</code> <code>a >= b</code> <code>a <=> b</code>	<code>a[b]</code> <code>*a</code> <code>&a</code> <code>a->b</code> <code>a.b</code> <code>a->*b</code> <code>a.*b</code>	<code>a(...)</code> <code>a, b</code> <code>? :</code>

Control structures

If statement

```
1 if (STATEMENT) {  
2     // This is executed if STATEMENT == true  
3 } else if (OTHER_STATEMENT) {  
4     // This is executed if:  
5     // (STATEMENT == false) && (OTHER_STATEMENT == true)  
6 } else {  
7     // This is executed if neither is true  
8 }
```

- Used to conditionally execute code
- All the `else` cases can be omitted if needed
- `STATEMENT` can be **any boolean expression**

Switch statement

```
1 switch(STATEMENT) {  
2     case CONST_1:  
3         // This runs if STATEMENT == CONST_1.  
4         break;  
5     case CONST_2:  
6         // This runs if STATEMENT == CONST_2.  
7         break;  
8     default:  
9         // This runs if no other options worked.  
10 }
```

- Used to conditionally execute code
- Can have many `case` statements
- `break` exits the `switch` block
- `STATEMENT` usually returns `int` or `enum` value

Switch statement, C style

```
1 #include <stdio.h>
2 int main() {
3     // Color could be:
4     // RED    == 1
5     // GREEN  == 2
6     // BLUE   == 3
7     int color = 2;
8     switch (color) {
9         case 1: printf("red\n"); break;
10        case 2: printf("green\n"); break;
11        case 3: printf("blue\n"); break;
12    }
13    return 0;
14 }
```

Switch statement, C++ style

```
1 #include <iostream>
2
3 int main() {
4     enum class RGB { RED, GREEN, BLUE };
5     RGB color = RGB::GREEN;
6
7     switch (color) {
8         case RGB::RED:    std::cout << "red\n"; break;
9         case RGB::GREEN: std::cout << "green\n"; break;
10        case RGB::BLUE:  std::cout << "blue\n"; break;
11    }
12    return 0;
13 }
```

While loop

```
1 while (STATEMENT) {  
2     // Loop while STATEMENT == true.  
3 }
```

Example `while` loop:

```
1 bool condition = true;  
2 while (condition) {  
3     condition = /* Magically update condition. */  
4 }
```

- Usually used when the exact number of iterations is unknown before-wise
- Easy to form an endless loop by mistake

For loop

```
1 for (INITIAL_CONDITION; END_CONDITION; INCREMENT) {  
2     // This happens until END_CONDITION == false  
3 }
```

Example `for` loop:

```
1 for (int i = 0; i < COUNT; ++i) {  
2     // This happens COUNT times.  
3 }
```

- In C++ `for` loops are *very* fast. Use them!
- Less flexible than `while` but less error-prone
- Use `for` when number of iterations is fixed and `while` otherwise

Range for loop

- Iterating over a standard containers like `array` or `vector` has simpler syntax
- Avoid mistakes with indices
- Show intent with the syntax
- Has been added in C++ 11

```
1 for (const auto& value : container) {  
2     // This happens for each value in the container.  
3 }
```

Spoiler Alert

New in C++ 17

```
1 std::map<char, int> my_dict{{'a', 27}, {'b', 3}};
2 for (const auto& [key, value] : my_dict) {
3     cout << key << " has value " << value << endl;
4 }
```

Similar to

```
1 my_dict = {'a': 27, 'b': 3}
2 for key, value in my_dict.items():
3     print(key, "has value", value)
```

Spoiler Alert 2

The C++ is ≈ 15 times faster than Python

```
/tmp/map_bench ./main.cpp
benchmarking ./main.cpp
time           1.971 ms  (1.882 ms .. 2.151 ms)
  0.998 R2  (0.997 R2 .. 0.999 R2)
mean           2.087 ms  (2.031 ms .. 2.178 ms)
std dev        237.9 µs  (158.4 µs .. 409.1 µs)
variance introduced by outliers: 74% (severely inflated)
```

```
/tmp/map_bench ./main.py
benchmarking ./main.py
time           32.71 ms  (31.41 ms .. 34.14 ms)
  0.995 R2  (0.992 R2 .. 0.999 R2)
mean           32.31 ms  (31.79 ms .. 33.08 ms)
std dev        1.312 ms  (871.7 µs .. 1.999 ms)
variance introduced by outliers: 11% (moderately inflated)
```

Exit loops and iterations

- We have control over loop iterations
- Use `break` to exit the loop
- Use `continue` to skip to next iteration

```
1 while (true) {  
2     int i = /* Magically get new int. */  
3     if (i % 2 == 0) {  
4         cerr << i << endl;  
5     } else {  
6         break;  
7     }  
8 }
```

Built-in types

Built-in types

“Out of the box” types in C++:

```
1 bool this_is_fun = true;      // Boolean: true or false.
2 char caret_return = '\n';    // Single character.
3 int meaning_of_life = 42;    // Integer number.
4 short smaller_int = 42;      // Short number.
5 long bigger_int = 42;        // Long number.
6 float fraction = 0.01f;      // Single precision float.
7 double precise_num = 0.01;   // Double precision float.
8 auto some_int = 13;          // Automatic type [int].
9 auto some_float = 13.0f;     // Automatic type [float].
10 auto some_double = 13.0;     // Automatic type [double].
```

[Reference]

<http://en.cppreference.com/w/cpp/language/types>

Operations on arithmetic types

- All **character**, **integer** and **floating point** types are arithmetic
- Arithmetic operations: `+`, `-`, `*`, `/`
- Comparisons `<`, `>`, `<=`, `>=`, `==` return `bool`
- `a += 1` \Leftrightarrow `a = a + 1`, same for `-=`, `*=`, `/=`, etc.
- Avoid `==` for floating point types

[Reference]

https://en.cppreference.com/w/cpp/language/arithmetic_types

Are we crazy?

```
1 #include <iostream>
2 int main() {
3     // Create an innocent float variable
4     const float var = 84.78;
5
6     // Let's compare the same number, they should be the
7     // same...
8     const bool cmp_result = (84.78 == var);
9     std::cout << "84.78 equal to " << var << "???\\n"
10    << std::boolalpha << cmp_result << '\\n';
11 }
```

true or false ???

Some additional operations

- Boolean variables have logical operations
or: `||`, **and**: `&&`, **not**: `!`

```
1 bool is_happy = (!is_hungry && is_warm) || is_rich
```

- Additional operations on integer variables:
 - `/` is integer division: i.e. `7 / 3 == 2`
 - `%` is modulo division: i.e. `7 % 3 == 1`
 - **Increment** operator: `a++` \Leftrightarrow `++a` \Leftrightarrow `a += 1`
 - **Decrement** operator: `a--` \Leftrightarrow `--a` \Leftrightarrow `a -= 1`
 - Do not use de- increment operators within another expression, i.e. `a = (a++) + ++b`



Coding Horror image from Code Complete 2 book by Steve McConnell

Variables

Declaring variables

Variable declaration always follows pattern:

<TYPE> <NAME> [= <VALUE>];

- Every variable has a type
- Variables cannot change their type
- **Always initialize** variables if you can

```
1 bool sad_uninitialized_var;  
2 bool initializing_is_good = true;
```

Naming variables

- Name **must** start with a letter
- Give variables **meaningful names**
- Don't be afraid to **use longer names**
- **Don't include type** in the name
- **Don't use negation** in the name
- **GOOGLE-STYLE** name variables in **`snake_case`**
all lowercase, underscores separate words
- C++ is case sensitive:
`some_var` is different from `some_Var`

Variables live in scopes

- There is a single global scope
- Local scopes start with `{` and ends with `}`
- All variables **belong to the scope** where they have been declared
- All variables die in the end of **their** scope
- This is the core of C++ memory system

```
1 int main() { // Start of main scope.
2     float some_float = 13.13f; // Create variable.
3     { // New inner scope.
4         auto another_float = some_float; // Copy variable.
5     } // another_float dies.
6     return 0;
7 } // some_float dies.
```

Any variable can be `const`

- Use `const` to declare a **constant**
- The compiler will guard it from any changes
- Keyword `const` can be used with **any** type
- **GOOGLE-STYLE** name constants in **CamelCase** starting with a small letter **k**:
 - `const float kImportantFloat = 20.0f;`
 - `const int kSomeInt = 20;`
 - `const std::string kHello = "hello";`
- `const` is part of type:
variable `kSomeInt` has type `const int`
- **Tip:** declare everything `const` unless it **must** be changed

References to variables

- We can create a **reference** to any variable
- Use **&** to state that a variable is a reference
 - `float& ref = original_variable;`
 - `std::string& hello_ref = hello;`
- Reference is part of type:
variable `ref` has type `float&`
- Whatever happens to a reference happens to the variable and vice versa
- Yields performance gain as references
avoid copying data

Const with references

- References are fast but reduce control
- To avoid unwanted changes use `const`
 - `const float& ref = original_variable;`
 - `const std::string& hello_ref = hello;`

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int num = 42;    // Name has to fit on slides
5     int& ref = num;
6     const int& kRef = num;
7     ref = 0;
8     cout << ref << " " << num << " " << kRef << endl;
9     num = 42;
10    cout << ref << " " << num << " " << kRef << endl;
11    return 0;
12 }
```

Streams

I/O streams (Lecture 0)

- Handle `stdin`, `stdout` and `stderr`:
 - `std::cin` — maps to `stdin`
 - `std::cout` — maps to `stdout`
 - `std::cerr` — maps to `stderr`
- `#include <iostream>` to use I/O streams
- Part of C++ standard library

```
1 #include <iostream>
2 int main() {
3     int some_number;
4     std::cout << "please input any number" << std::endl;
5     std::cin >> some_number;
6     std::cout << "number = " << some_number << std::endl;
7     std::cerr << "boring error message" << std::endl;
8     return 0;
9 }
```

What does this program do?

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5     char filename[] = "00205.txt";
6     char *pch;
7     pch = strtok(filename, ".");
8     while (pch != NULL) {
9         printf("%s\n", pch);
10        pch = strtok(NULL, ".");
11    }
12    return 0;
13 }
```

String streams

Already known streams:

- Standard output: `cerr`, `cout`
- Standard input: `cin`
- Filestreams: `fstream`, `ifstream`, `ofstream`

New type of stream: `stringstream`

- Combine `int`, `double`, `string`, etc. into a single `string`
- Break up `strings` into `int`, `double`, `string` etc.

```
1 #include <iomanip>
2 #include <iostream>
3 #include <sstream>
4 using namespace std;
5
6 int main() {
7     // Combine variables into a stringstream.
8     stringstream filename{"00205.txt"};
9
10    // Create variables to split the string stream
11    int num = 0;
12    string ext;
13
14    // Split the string stream using simple syntax
15    filename >> num >> ext;
16
17    // Tell your friends
18    cout << "Number is: " << num << endl;
19    cout << "Extension is: " << ext << endl;
20    return 0;
21 }
```

Program input parameters

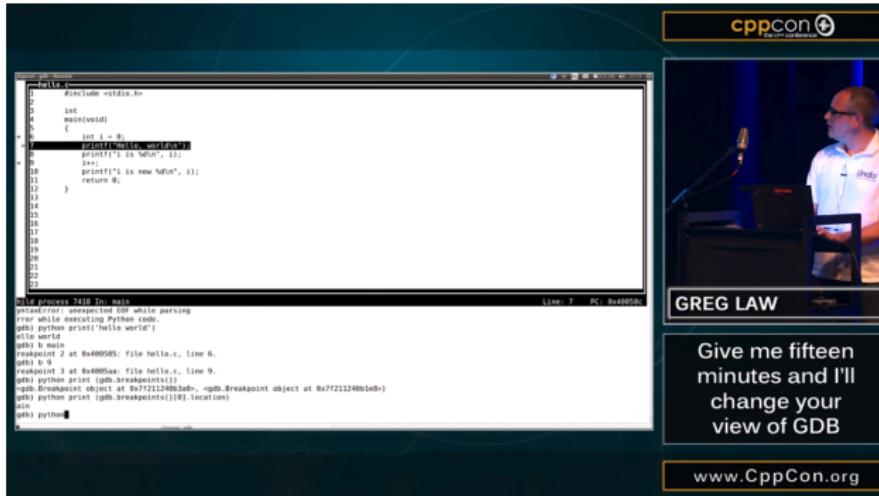
- Originate from the declaration of main function
- Allow passing arguments to the binary
- `int main(int argc, char const *argv[]);`
- `argc` defines number of input parameters
- `argv` is an array of string parameters
- By default:
 - `argc == 1`
 - `argv == "<binary_path>"`

Program input parameters

```
1 #include <iostream>
2 #include <string>
3 using std::cout;
4 using std::endl;
5
6 int main(int argc, char const *argv[]) {
7     // Print how many parameters we received
8     cout << "Got " << argc << " params\n";
9
10    // First program argument is always the program name
11    cout << "Program: " << argv[0] << endl;
12
13    for (int i = 1; i < argc; ++i) { // from 1 on
14        cout << "Param: " << argv[i] << endl;
15    }
16    return 0;
17 }
```

Suggested Video

“Give me 15 minutes & I’ll change your view of GDB”



<https://youtu.be/PorfLSr3DDI>

References

C++ language

This is a reference of the core C++ language constructs.

Basic concepts

Comments
ASCII chart
Names and identifiers
Types - Fundamental types
Object - Scope - Lifetime
Definitions and ODR
Name lookup
qualified - unqualified
As-if rule
Undefined behavior
Memory model and data races
Phases of translation
The main() function
Modules([C++20](#))

C++ Keywords

Preprocessor

```
#if - #ifdef - #else - #endif  
#define - # - ## - #include  
#error - #pragma - #line
```

Expressions

Value categories
Evaluation order and sequencing
Constant expressions
Operators
assignment - arithmetic
increment and decrement
logical - comparison
member access and indirection
call, comma, ternary
sizeof, alignof([C++11](#))
new - delete - typeid
Operator overloading
Default comparisons([C++20](#))
Operator precedence
Conversion operators
implicit - explicit - user-defined
static cast - dynamic cast
const_cast - reinterpret_cast
Literals
boolean - integer - floating
character - string
nullptr([C++11](#))
user-defined ([C++11](#))

Declaration

Namespace declaration
Namespace alias
Lvalue and rvalue references
Pointers - Arrays
Structured bindings([C++17](#))
Enumerations and enumerators
Storage duration and linkage
Language linkage
inline specifier
inline assembly
const/volatile
constexpr([C++11](#))
constexpr([C++20](#)) - constinit([C++20](#))
decltype([C++11](#)) - auto([C++11](#))
alignas([C++11](#))
typedef - Type alias([C++11](#))
Elaborated type specifiers
Attributes([C++11](#))
static_assert([C++11](#))

Initialization

Default initialization
Value initialization([C++03](#))
Copy initialization
Direct initialization
Aggregate initialization
List initialization([C++11](#))
Reference initialization
Static non-local initialization
zero - constant
Dynamic non-local initialization
ordered - unordered
Copy elision

Functions

Function declaration
Default arguments
Variable arguments
Lambda expressions([C++11](#))
Argument-dependent lookup
Overload resolution
Operator overloading
Address of an overload set
Coroutines ([C++20](#))

Statements

```
if - switch  
for - range-for(C++11)  
while - do-while  
continue - break - goto - return  
synchronized and atomic(TM TS)
```

Classes

Class types - Union types
Injected-class-name
Data members - Member functions
Static members - Nested classes
Derived class - using-declaration
Virtual functions - Abstract class
override([C++11](#)) - final([C++11](#))
Member access - friend
Bit fields - The this pointer
Constructors and member initializer lists
Default constructor - Destructor
Copy constructor - Copy assignment
Move constructor([C++11](#))
Move assignment([C++11](#))
Converting constructor - explicit specifier

Templates

Template parameters and arguments
Class template - Function template
Class member template
Variable template([C++14](#))
Template argument deduction
Explicit specialization
Class template argument deduction([C++17](#))
Partial specialization
Parameter packsize([C++11](#)) - sizeof...([C++11](#))
Fold-expressions([C++17](#))
Dependent names - SFINAE
Constraints and concepts ([C++20](#))

Exceptions

throw-expression
try-catch block
function-try-block
noexcept specifier([C++11](#))
noexcept operation([C++11](#))
Dynamic exception specification([Until C++17](#))

Miscellaneous

History of C++
Extending the namespace std
Acronyms

Idioms

Resource acquisition is initialization
Rule of three/five/zero
Pointer to implementation

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

Modern C++ for Computer Vision and Image Processing

Lecture 3: C++ Functions

Ignacio Vizzo and Cyrill Stachniss

C-style strings are evil

Like everything else in C in general.

```
1 #include <cstring>
2 #include <iostream>
3
4 int main() {
5     const char source[] = "Copy this!";
6     char dest[5];
7     std::cout << source << '\n';
8
9     std::strcpy(dest, source);
10    std::cout << dest << '\n';
11
12    // source is const, no problem right?
13    std::cout << source << '\n';
14
15    return 0;
16 }
```

Strings

- `#include <string>` to use `std::string`
- Concatenate strings with `+`
- Check if `str` is empty with `str.empty()`
- Works out of the box with I/O streams

```
1 #include <iostream>
2 #include <string>
3
4 int main() {
5     const std::string source{"Copy this!"};
6     std::string dest = source; // copy string
7
8     std::cout << source << '\n';
9     std::cout << dest << '\n';
10    return 0;
11 }
```

Function definition

*In programming, a named section of a program that performs a **specific** task. In this sense, a **function** is a type of **procedure** or **routine**. Some programming languages make a distinction between a **function**, which returns a value, and a **procedure**, which performs some operation but does not return a value.*

Bjarne Stroustrup

*The main way of getting something done in a C++ program is to call a **function** to do it. Defining a **function** is the way you specify how an operation is to be done. A **function** cannot be called unless it has been previously declared. A **function** declaration gives the name of the **function**, the type of the value returned (if any), and the number and types of the arguments that must be supplied in a call.*

Functions

```
1 ReturnType FuncName(ParamType1 in_1, ParamType2 in_2) {  
2     // Some awesome code here.  
3     return return_value;  
4 }
```

- 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**

Function Anatomy

```
1 [[attributes]] ReturnType FuncName(ArgumentList...) {  
2     // Some awesome code here.  
3     return return_value;  
4 }
```

- Body
- Optional Attributes
- Return Type
- Name
- Argument List

Funtcion Body

- Where the **computation** happens.
- Defines a new scope, the **scope of the function**.
- Access outside world(scopes) through input arguments.
- Can not add information about the implementation outside this **scope**

Funtcion Body

```
1 // This is not part of the body of the function
2
3 void MyFunction() {
4     // This is the body of the function
5     // Whatever is inside here is part of
6     // the scope of the function
7 }
8
9 // This is not part of the body of the function
```

Return Type

Could be any of:

1. An **unique** type, eg: `int`, `std::string`, etc...
2. `void`, also called subroutine.

Rules:

- If has a return type, **must** return a value.
- If returns void, must **NOT** return any value.

Return Type

```
1 int f1() {} // error
2 void f2() {} // OK
3
4 int f3() { return 1; } // error
5 void f4() { return 1; } // OK
6
7 int f5() { return; } // error
8 void f6() { return; } // OK
```

Return Type

Automatic return type deduction C++14):

```
1 std::map<char, int> GetDictionary() {  
2     return std::map<char, int>{{'a', 27}, {'b', 3}};  
3 }
```

Can be expressed as:

```
1 auto GetDictionary() {  
2     return std::map<char, int>{{'a', 27}, {'b', 3}};  
3 }
```

Return Type

Sadly you can use only one type for return values, so, no **Python** like:

```
1 #!/usr/bin/env python3
2 def foo():
3     return "Super Variable", 5
4
5
6 name, value = foo()
7 print(name + " has value: " + str(value))
```

Return Type

Sadly you can use only one type for return values, so no Python like:

```
1 #!/usr/bin/env python3
2 def foo():
3     return "Super Variable", 5
4
5
6 name, value = foo()
7 print(name + " has value: " + str(value))
```

Let's write this in C++, and make it run **18** times faster, with a similar syntax.

Return Type

With the introduction of structured binding in C++17 you can now:

```
1 #include <iostream>
2 #include <tuple>
3 using namespace std;
4
5 auto Foo() {
6     return make_tuple("Super Variable", 5);
7 }
8
9 int main() {
10    auto [name, value] = Foo();
11    cout << name << " has value :" << value << endl;
12    return 0;
13 }
```

Return Type

WARNING:

Never return reference to locally variables!!!

```
1 #include <iostream>
2 using namespace std;
3
4 int& MultiplyBy10(int num) { // retval is created
5     int retval = 0;
6     retval = 10 * num;
7     return retval;
8 } // retval is destroyed, it's not accessible anymore
9
10 int main() {
11     int out = MultiplyBy10(10);
12     cout << "out is " << out << endl;
13     return 0;
14 }
```

Return Type

```
1 #include <iostream>
2 using namespace std;
3
4 int& MultiplyBy10(int num) { // retval is created
5     int retval = 0;
6     retval = 10 * num;
7     cout << "retval is " << retval << endl;
8     return retval;
9 } // retval is destroyed, it's not accessible anymore
10
11 int main() {
12     int out = MultiplyBy10(10);
13     cout << "out      is " << out << endl;
14     return 0;
15 }
```

Return Type

Compiler got your back:

Return value optimization:

https://en.wikipedia.org/wiki/Copy_elision#Return_value_optimization

```
1 Type DoSomething() {  
2     Type huge_variable;  
3  
4     // ... do something  
5  
6     // don't worry, the compiler will optimize it  
7     return huge_variable;  
8 }  
9  
10 // ...  
11  
12 Type out = DoSomething(); // does not copy
```

Local Variables

- A local variable is initialized when the execution reaches its definition.

```
1 void f() {  
2     // Gets initialized when the execution reaches  
3     // the definition of f(), thus, this implementation  
4     int local_variable = 50;  
5 }  
6  
7 // at this point local_variable has not been initialized  
8 // is not accessible by any other part of the program  
9  
10 f(); //< When enter the function call, gets initialized
```

Local Variables

- Unless declared `static`, each invocation has its own copy.

```
1 void f() {  
2     float var1 = 5.5F;  
3     float var2 = 1.5F;  
4     // do something with var1, var2  
5 }  
6  
7 f(); //< First call, var1, var2 are created  
8 f(); //< Second call, NEW var1, var2 are created
```

Local Variables

- **static** variable, a single, statically allocated object represent that variable in **all** calls.

```
1 void f() {  
2     // same variable for all function calls  
3     static int counter = 0;  
4  
5     // Increment counter on each function call  
6     counter++;  
7 }  
8  
9 // at this point, f::counter has been statically  
10 // allocated and accessible by any function call to f()  
11  
12 f(); //< Acess counter, counter == 1  
13 f(); //< Acess same counter, counter ==2
```

Local Variables

```
1 #include <iostream>
2 using namespace std;
3 void Counter() {
4     static int counter = 0;
5     cout << "counter state = " << ++counter << endl;
6 }
7 int main() {
8     for (size_t i = 0; i < 5; i++) {
9         Counter();
10    }
11    return 0;
12 }
```

NACHO-STYLE Avoid if possible, read more at:

<https://isocpp.org/wiki/faq/ctors#static-init-order>

Local Variables

- Any local variable will be destroyed when the execution exit the `scope` of the function.

```
1 void f() {  
2     // Gets initialized when the execution reaches  
3     // the defintion of f(), thus, this implementation  
4     int local_variable = 50;  
5 }
```

`local_variable` has been destroyed at this point, RIP.

Argument List

- **How** the function interact with external world
- They all have a **type**, and a **name** as well.
- They are also called **parameters**.
- Unless is declared as reference, a **copy** of the actual argument is passed to the function.

Argument List

```
1 void f(type arg1, type arg2) {  
2     // f holds a copy of arg1 and arg2  
3 }  
4  
5 void f(type& arg1, type& arg2) {  
6     // f holds a reference of arg1 and arg2  
7     // f could possibly change the content  
8     // of arg1 or arg2  
9 }  
10  
11 void f(const type& arg1, const type& arg2) {  
12     // f can't change the content of arg1 nor arg2  
13 }  
14  
15 void f(type arg1, type& arg2, const type& arg3);
```

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
- **NACHO-STYLE** Never use them

Example: default arguments

```
1 #include <iostream>
2 using namespace std;
3
4 string SayHello(const string& to_whom = "world") {
5     return "Hello " + to_whom + "!";
6 }
7
8 int main() {
9     // Will call SayHello using the default argument
10    cout << SayHello() << endl;
11
12    // This will override the default argument
13    cout << SayHello("students") << endl;
14    return 0;
15 }
```

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

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



CODING HORROR

Is the string still the same?

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

Unknown without looking into `DoSmth()`!

Solution: use const references

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

```
1 void DoSmth(const std::string& huge_string);
```

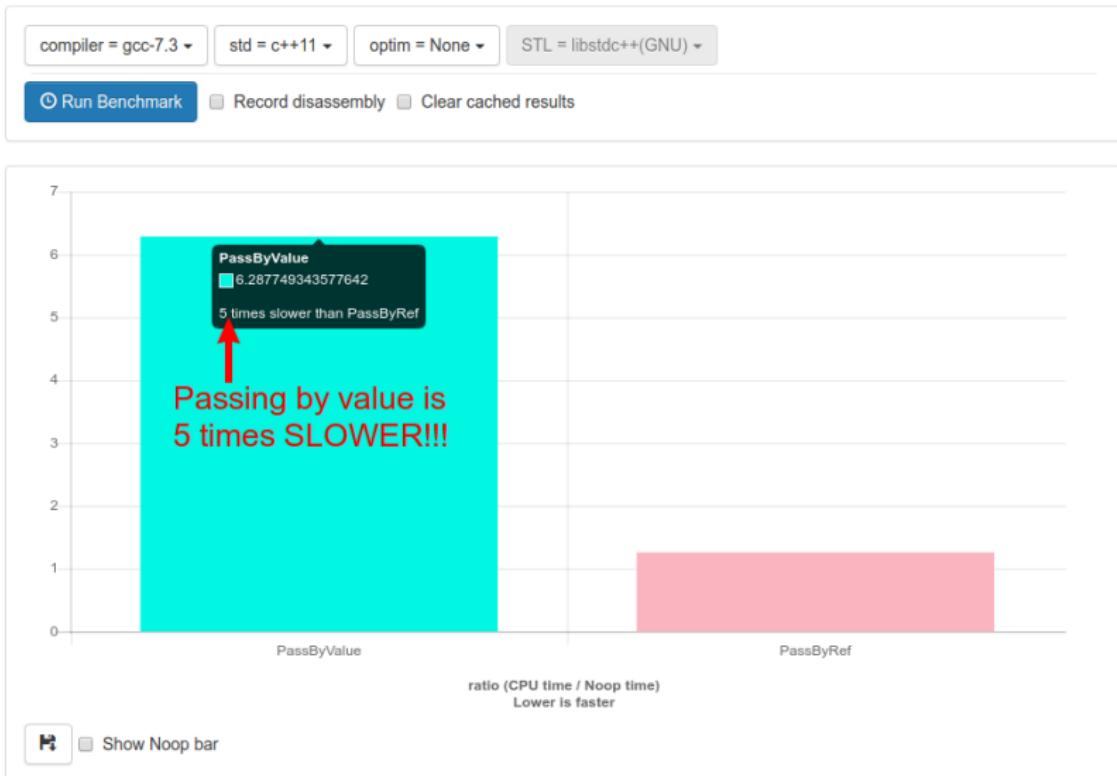
- Use **snake_case** for all function arguments
- Non-const refs are mostly used in older code written before C++ 11
- They can be useful but destroy readability
- **GOOGLE-STYLE** Avoid using non-const refs

Cost of passing by value

```
1 void pass_by_value(std::string huge_string) {  
2     (void) huge_string; ←  
3 }  
4  
5 // Pay attention to the -> "&" <- symbol  
6 void pass_by_ref(std::string& huge_string) {  
7     (void) huge_string; ←  
8 }  
9  
10 static void PassByValue(benchmark::State& state) {  
11     // Code inside this loop is measured repeatedly  
12     std::string created_string("hello");  
13     for (auto _ : state) {  
14         pass_by_value(created_string); ←  
15     }  
16 }  
17 BENCHMARK(PassByValue);  
18  
19 static void PassByRef(benchmark::State& state) {  
20     // Code inside this loop is measured repeatedly  
21     std::string created_string("hello");  
22     for (auto _ : state) {  
23         pass_by_ref(created_string);  
24     }  
25 }  
26 BENCHMARK(PassByRef); ←  
    This function receive a copy  
    of the value of the input string.  
    If the input string is big, this  
    operation might cost a lot of time  
    This function receive a reference  
    to the input_string. We will access  
    the memory location where the  
    input string is located  
    This function call  
    will be evaluated  
    in the benchmark  
    Pay attention to the  
    benchmark names
```

<http://quick-bench.com/LqwB1COM3KrQE4tqupBtzqJmCdw>

Cost of passing by value



inline

- `function` calls are expensive...
- Well, not **THAT** expensive though.
- If the function is rather small, you could help the compiler.
- `inline` is a **hint** to the compiler
 - should attempt to generate code for a call
 - rather than a function call.
- Sometimes the compiler do it anyways.

inline

```
1 inline int fac(int n) {
2     if (n < 2) {
3         return 2;
4     }
5     return n * fac(n - 1);
6 }
7
8 int main() { return fac(10); }
```

Check it out:

<https://godbolt.org/z/amkfH4>

```
1 inline int fac(int n) {
2     if (n < 2) {
3         return 2;
4     }
5     return n * fac(n - 1);
6 }
7
8 int main() {
9     int fac0 = fac(0);
10    int fac1 = fac(1);
11    int fac2 = fac(2);
12    int fac3 = fac(3);
13    int fac4 = fac(4);
14    int fac5 = fac(5);
15    return fac0 + fac1 + fac2 + fac3 + fac4 + fac5;
16 }
```

Check it out:

<https://godbolt.org/z/EGd6aG>

C-style overloading

cosine

```
1 #include <math.h>
2
3 double cos(double x);
4 float cosf(float x);
5 long double cosl(long double x);
```

arctan

```
1 #include <math.h>
2
3 double atan(double x);
4 float atanf(float x);
5 long double atanl(long double x);
```

C-style overloading

usage

```
1 #include <math.h>
2 #include <stdio.h>
3
4 int main() {
5     double x_double = 0.0;
6     float x_float = 0.0;
7     long double x_long_double = 0.0;
8
9     printf("cos(0) = %f\n", cos(x_double));
10    printf("cos(0) = %f\n", cosf(x_float));
11    printf("cos(0) = %Lf\n", cosl(x_long_double));
12
13    return 0;
14 }
```

C++ style overloading

cosine

```
1 #include <cmath>
2
3 // ONE cos function to rule them all
4 double cos(double x);
5 float cos(float x);
6 long double cos(long double x);
```

arctan

```
1 #include <cmath>
2
3 double atan(double x);
4 float atan(float x);
5 long double atan(long double x);
```

C++ style overloading

usage

```
1 #include <cmath>
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     double x_double = 0.0;
7     float x_float = 0.0;
8     long double x_long_double = 0.0;
9
10    cout << "cos(0)=" << std::cos(x_double) << '\n';
11    cout << "cos(0)=" << std::cos(x_float) << '\n';
12    cout << "cos(0)=" << std::cos(x_long_double) << '\n';
13
14    return 0;
15 }
```

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

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4 string TypeOf(int) { return "int"; }
5 string TypeOf(const string&) { return "string"; }
6 int main() {
7     cout << TypeOf(1) << endl;
8     cout << TypeOf("hello") << endl;
9     return 0;
10 }
```

Good Practices

- Break up complicated computations into meaningful chunks and name them.
- Keep the length of functions **small** enough.
- Avoid **unnecessary** comments.
- One function should achieve **ONE** task.
- If you can't pick a short name, then **split** functionality.
- Avoid **macros**.
 - If you must use it, use ugly names with lots of capital letters.

Good function example

```
1 #include <vector>
2 using namespace std;
3
4 vector<int> CreateVectorOfZeros(int size) {
5     vector<int> null_vector(size);
6     for (int i = 0; i < size; ++i) {
7         null_vector[i] = 0;
8     }
9     return null_vector;
10}
11
12 int main() {
13     vector<int> zeros = CreateVectorOfZeros(10);
14     return 0;
15 }
```

Bad function example #1

```
1 #include <vector>
2 using namespace std;
3 vector<int> Func(int a, bool b) {
4     if (b) { return vector<int>(10, a); }
5     vector<int> vec(a);
6     for (int i = 0; i < a; ++i) { vec[i] = a * i; }
7     if (vec.size() > a * 2) { vec[a] /= 2.0f; }
8     return vec;
9 }
```



- Name of the function means nothing
- Names of variables mean nothing
- Function does not have a single purpose

Bad function example #2

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 vector<int> CreateVectorAndPrintContent(int size) {
6     vector<int> vec(size);
7     for (size_t i = 0; i < size; i++) {
8         vec[i] = 0;
9         cout << vec[i] << endl;
10    }
11    return vec;
12 }
13
14 int main() {
15     vector<int> zeros = CreateVectorAndPrintContent(5);
16     return 0;
17 }
```

Bad function example #2 fix

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 vector<int> CreateVector(int size) {
6     vector<int> vec(size);
7     for (size_t i = 0; i < size; i++) {
8         vec[i] = 0;
9     }
10    return vec;
11 }
12
13 void PrintVector(std::vector<int> vec) {
14     for (auto element : vec) {
15         cout << element << endl;
16     }
17 }
```

Bad function example #3

```
1 // function user will only see the declaration
2 // and NOT the definition.
3 // It's impossible to know at this point any
4 // additional information about this function.
5 int SquareNumber(int num);
```

Bad function example #3



Bad function example #3

```
1 // function user will only see the declaration
2 // and NOT the definition.
3 // It's impossible to know at this point any
4 // additional information about this function.
5 int SquareNumber(int num);
```

```
1 int SquareNumber(int num) {
2     // by the way, you need to call your aunt
3     // at this point. otherwise the program will
4     // fail with error code 314.
5     CallYourAunt();
6
7     return num * num;
8 }
```

Namespaces

module1

```
namespace module_1 {  
    int SomeFunc() {}  
}
```

module2

```
namespace module_2 {  
    int SomeFunc() {}  
}
```

- Helps avoiding name conflicts
- Group the project into logical modules

Namespaces example

```
1 #include <iostream>
2
3 namespace fun {
4     int GetMeaningOfLife(void) { return 42; }
5 } // namespace fun
6
7 namespace boring {
8     int GetMeaningOfLife(void) { return 0; }
9 } // namespace boring
10
11 int main() {
12     std::cout << boring::GetMeaningOfLife() << std::endl
13             << fun::GetMeaningOfLife() << std::endl;
14     return 0;
15 }
```

Namespaces example 2

- We don't like `std::vector` at all
- Let's define our own vector Class

```
1 // @file: my_vector.hpp
2 namespace my_vec {
3     template <typename T>
4     class vector {
5         // ...
6     };
7 } // namespace my_vec
```

Namespaces example 2

```
1 #include <vector>
2 #include "my_vecor.hpp"
3 int main() {
4     std::vector<int> v1;      // Standard vector.
5     vec::vector<int> v2;      // User defined vector.
6
7     {
8         using std::vector;
9         vector<int> v3;      // Same as std::vector
10        v1 = v3;             // OK
11    }
12
13    {
14        using vec::vector;
15        vector<int> v4;      // Same as vec::vector
16        v2 = v4;             // OK
17    }
18 }
```

Avoid using namespace <name>

```
1 #include <cmath>
2 #include <iostream>
3 using namespace std; // std namespace is used
4
5 // Self-defined function power shadows std::pow
6 double pow(double x, int exp) {
7     double res = 1.0;
8     for (int i = 0; i < exp; i++) {
9         res *= x;
10    }
11    return res;
12 }
13
14 int main() {
15     cout << "2.0^2 = " << pow(2.0, 2) << endl;
16     return 0;
17 }
```

Namespace error

Error output:

```
1 /home/ivizzo/.../namespaces_error.cpp:13:26:  
2 error: call of overloaded 'pow(double&, int&)' is  
      ambiguous  
3 double res = pow(x, exp);  
4  
5 ...
```

Only use what you need

```
1 #include <cmath>
2 #include <iostream>
3 using std::cout; // Explicitly use cout.
4 using std::endl; // Explicitly use endl.
5
6 // Self-defined function power shadows std::pow
7 double pow(double x, int exp) {
8     double res = 1.0;
9     for (int i = 0; i < exp; i++) {
10         res *= x;
11     }
12     return res;
13 }
14
15 int main() {
16     cout << "2.0^2 = " << pow(2.0, 2) << endl;
17     return 0;
18 }
```

Namespaces Wrap Up

Use namespaces to avoid name conflicts

```
1 namespace some_name {  
2     <your_code>  
3 } // namespace some_name
```

Use using correctly

- **[good]**
 - `using my_namespace::myFunc;`
 - `my_namespace::myFunc(...);`
- **Never** use `using namespace` name in `*.hpp` files
- Prefer using explicit `using` even in `*.cpp` files

Nameless namespaces

- GOOGLE-STYLE for namespaces:

<https://google.github.io/styleguide/cppguide.html#Namespaces>

- GOOGLE-STYLE If you find yourself relying on some constants in a file and these constants should not be seen in any other file, put them into a **nameless namespace** on the top of this file

```
1 namespace {  
2     const int kLocalImportantInt = 13;  
3     const float kLocalImportantFloat = 13.0f;  
4 } // namespace
```

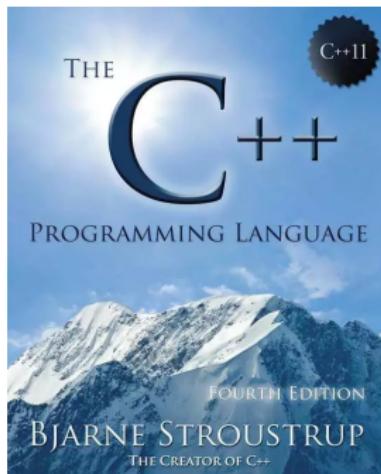
https://google.github.io/styleguide/cppguide.html#Unnamed_Namespaces_and_Static_Variables

Suggested Video



<https://youtu.be/cVC8bcV8zsQ>

References



■ Website:

<http://www.stroustrup.com/4th.html>

References

- **Functions**

Stroustrup's book, chapter 12

- **Namespaces**

Stroustrup's book, chapter 14

- **cppreference**

<https://en.cppreference.com/w/cpp/language/function>

- **c-style strings**

<https://www.learncpp.com/cpp-tutorial/66-c-style-strings/>

Modern C++ for Computer Vision and Image Processing

Lecture 04: C++ STL Library

Ignacio Vizzo and Cyrill Stachniss

std::array

```
1 #include <array>
2 #include <iostream>
3 using std::cout;
4 using std::endl;
5
6 int main() {
7     std::array<float, 3> data{10.0F, 100.0F, 1000.0F};
8
9     for (const auto& elem : data) {
10         cout << elem << endl;
11     }
12
13     cout << std::boolalpha;
14     cout << "Array empty: " << data.empty() << endl;
15     cout << "Array size : " << data.size() << endl;
16 }
```

std::array

- `#include <array>` to use `std::array`
- Store a **collection of items** of **same type**
- Create from data:
`array<float, 3> arr = {1.0f, 2.0f, 3.0f};`
- Access items with `arr[i]`
indexing starts with **0**
- Number of stored items: `arr.size()`
- Useful access aliases:
 - First item: `arr.front() == arr[0]`
 - Last item: `arr.back() == arr[arr.size() - 1]`

std::vector

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 using std::cout;
5 using std::endl;
6
7 int main() {
8     std::vector<int> numbers = {1, 2, 3};
9     std::vector<std::string> names = {"Nacho", "Cyrill"};
10
11     names.emplace_back("Roberto");
12
13     cout << "First name : " << names.front() << endl;
14     cout << "Last number: " << numbers.back() << endl;
15     return 0;
16 }
```

std::vector

- `#include <vector>` to use `std::vector`
- Vector is implemented as a **dynamic table**
- Access stored items just like in `std::array`
- Remove all elements: `vec.clear()`
- Add a new item in one of two ways:
 - `vec.emplace_back(value)` [preferred, c++11]
 - `vec.push_back(value)` [historically better known]
- **Use it! It is fast and flexible!**

Consider it to be a default container to store collections of items of any same type

Optimize vector resizing

- `std::vector` size unknown.
- Therefore a `capacity` is defined.
- `size ≠ capacity`
- Many `push_back/emplace_back` operations force vector to change its `capacity` many times
- `reserve(n)` ensures that the vector has enough memory to store `n` items
- The parameter `n` can even be approximate
- This is a very **important optimization**

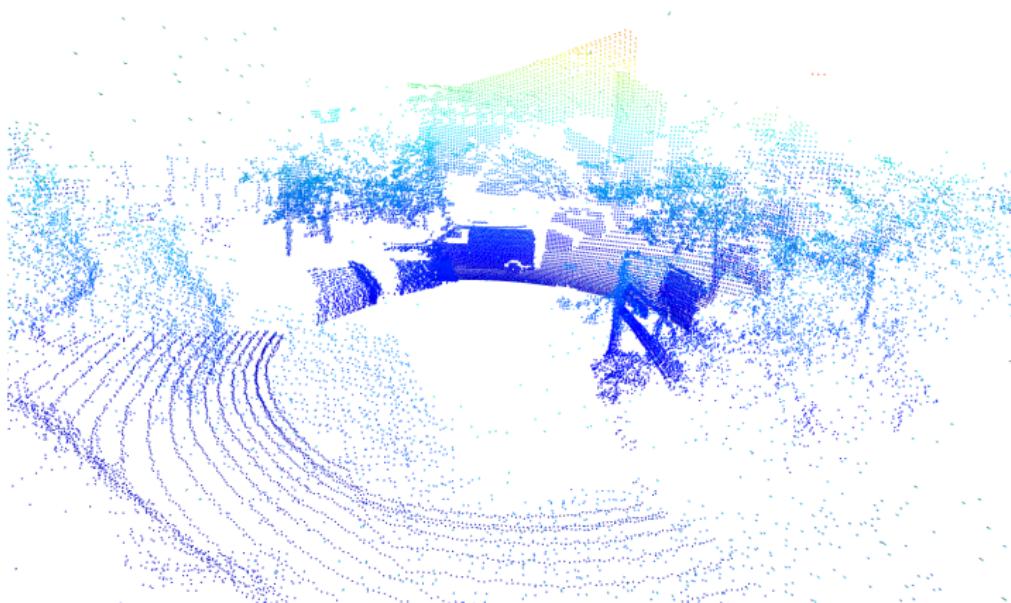
Optimize vector resizing

```
1 int main() {
2     const int N = 100;
3
4     vector<int> vec;    // size 0, capacity 0
5     vec.reserve(N);    // size 0, capacity 100
6     for (int i = 0; i < N; ++i) {
7         vec.emplace_back(i);
8     }
9     // vec ends with size 100, capacity 100
10
11    vector<int> vec2;    // size 0, capacity 0
12    for (int i = 0; i < N; ++i) {
13        vec2.emplace_back(i);
14    }
15    // vec2 ends with size 100, capacity 128
16 }
```

Containers in CV

Open3D::PointCloud

```
1 std::vector<Eigen::Vector3d> points_;  
2 std::vector<Eigen::Vector3d> normals_;  
3 std::vector<Eigen::Vector3d> colors_;
```



Size of container

sizeof()

```
1 int data[17];
2 size_t data_size = sizeof(data) / sizeof(data[0]);
3 printf("Size of array: %zu\n", data_size);
```

size()

```
1 std::array<int, 17> data_{};
2 cout << "Size of array: " << data_.size() << endl;
```

Empty Container

No standard way of checking if empty

```
1 int empty_arr[10];
2 printf("Array empty: %d\n", empty_arr[0] == NULL);
3
4 int full_arr[5] = {1, 2, 3, 4, 5};
5 printf("Array empty: %d\n", full_arr[0] == NULL);
```

empty()

```
1 std::vector<int> empty_vec_{};
2 cout << "Array empty: " << empty_vec_.empty() << endl;
3
4 std::vector<int> full_vec_{1, 2, 3, 4, 5};
5 cout << "Array empty: " << full_vec_.empty() << endl;
```

Access last element

No robust way of doing it

```
1 float f_arr[N] = {1.5, 2.3};  
2 // is it 3, 2 or 900?  
3 printf("Last element: %f\n", f_arr[3]);
```

back()

```
1 std::array<float, 2> f_arr_{1.5, 2.3};  
2 cout << "Last Element: " << f_arr_.back() << endl;
```

Clear elements

External function call, doesn't always work with floating points

```
1 char letters[5] = {'n', 'a', 'c', 'h', 'o'};  
2 memset(letters, 0, sizeof(letters));
```

clear()

```
1 std::vector<char> letters_ = {'n', 'a', 'c', 'h', 'o'};  
2 letters_.clear();
```

Remember std::string

```
1 std::string letters_right_{"nacho"};  
2 letters_right_.clear();
```

Why containers?

- Why **Not**?
- Same speed as **C-style** arrays but safer.
- Code readability.
- More functionality provided than a plain **C-style** array:
 - `size()`
 - `empty()`
 - `front()`
 - `back()`
 - `swap()`
 - STL algorithms...
 - Much more!

Much more...

More information about `std::vector`

<https://en.cppreference.com/w/cpp/container/vector>

More information about `std::array`

<https://en.cppreference.com/w/cpp/container/arra>

std::map

- **sorted** associative container.
- Contains **key-value** pairs.
- **keys** are unique.
- **keys** are stored using the `<` operator.
 - Your **keys** should be comparable.
 - built-in types always work, eg: `int`, `float`, etc
 - We will learn how to make your own types “comparable”.
- **value** can be any type, you name it.
- This are called dictionaries `dict` in Python.

std::map

- Create from data:

```
1 std::map<KeyT, ValueT> m{{key1, value1}, {..}};
```

- Check size: `m.size()`;
- Add item to map: `m.emplace(key, value)`;
- Modify or add item: `m[key] = value`;
- Get (const) ref to an item: `m.at(key)`;
- Check if key present: `m.count(key) > 0`;
 - Starting in C++20:
 - Check if key present: `m.contains(key)` [bool]

```
1 #include <iostream>
2 #include <map>
3 using namespace std;
4
5 int main() {
6     using StudentList = std::map<int, string>;
7     StudentList cpp_students;
8
9     // Inserting data in the students dictionary
10    cpp_students.emplace(1509, "Nacho");      // [1]
11    cpp_students.emplace(1040, "Pepe");        // [0]
12    cpp_students.emplace(8820, "Marcelo");     // [2]
13
14    for (const auto& [id, name] : cpp_students) {
15        cout << "id: " << id << ", " << name << endl;
16    }
17
18    return 0;
19 }
```

std::unordered_map

- Serves same purpose as `std::map`
- Implemented as a **hash table**
- Key type has to be hashable
- Typically used with `int`, `string` as a key
- Exactly same interface as `std::map`
- Faster to use than `std::map`

```
1 #include <iostream>
2 #include <unordered_map>
3 using namespace std;
4
5 int main() {
6     using StudentList = std::unordered_map<int, string>;
7     StudentList cpp_students;
8
9     // Inserting data in the students dictionary
10    cpp_students.emplace(1509, "Nacho");      // [2]
11    cpp_students.emplace(1040, "Pepe");        // [1]
12    cpp_students.emplace(8820, "Marcelo");     // [0]
13
14    for (const auto& [id, name] : cpp_students) {
15        cout << "id: " << id << ", " << name << endl;
16    }
17
18    return 0;
19 }
```

```
1 #include <functional>
2 template<> struct hash<bool>;
3 template<> struct hash<char>;
4 template<> struct hash<signed char>;
5 template<> struct hash<unsigned char>;
6 template<> struct hash<char8_t>; // C++20
7 template<> struct hash<char16_t>;
8 template<> struct hash<char32_t>;
9 template<> struct hash<wchar_t>;
10 template<> struct hash<short>;
11 template<> struct hash<unsigned short>;
12 template<> struct hash<int>;
13 template<> struct hash<unsigned int>;
14 template<> struct hash<long>;
15 template<> struct hash<long long>;
16 template<> struct hash<unsigned long>;
17 template<> struct hash<unsigned long long>;
18 template<> struct hash<float>;
19 template<> struct hash<double>;
20 template<> struct hash<long double>;
21 template<> struct hash<std::nullptr_t>; // C++17
```

Iterating over maps

```
1 for (const auto& kv : m) {  
2     const auto& key = kv.first;  
3     const auto& value = kv.second;  
4     // Do important work.  
5 }
```

New in C++17

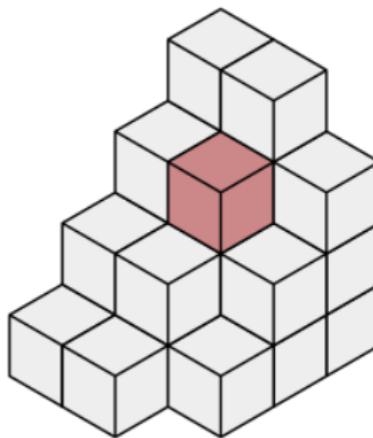
```
1 std::map<char, int> my_dict{{'a', 27}, {'b', 3}};  
2 for (const auto& [key, value] : my_dict) {  
3     cout << key << " has value " << value << endl;
```

- Every stored element is a pair
- `map` has keys **sorted**
- `unordered_map` has keys in **random** order

Associative Containers in CV

Open3D::VoxelGrid

```
1 std::unordered_map<Eigen::Vector3i ,  
2                     Voxel ,  
3                     hash_eigen::hash<Eigen::Vector3i>>  
4 voxels_;
```



Much more

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

array (C++11)	static contiguous array (class template)
vector	dynamic contiguous array (class template)
deque	double-ended queue (class template)
forward_list (C++11)	singly-linked list (class template)
list	doubly-linked list (class template)

Associative containers

Associative containers implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

set	collection of unique keys, sorted by keys (class template)
map	collection of key-value pairs, sorted by keys, keys are unique (class template)
multiset	collection of keys, sorted by keys (class template)
multimap	collection of key-value pairs, sorted by keys (class template)

Much more

Unordered associative containers

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ($O(1)$ amortized, $O(n)$ worst-case complexity).

<code>unordered_set</code> (C++11)	collection of unique keys, hashed by keys (class template)
<code>unordered_map</code> (C++11)	collection of key-value pairs, hashed by keys, keys are unique (class template)
<code>unordered_multiset</code> (C++11)	collection of keys, hashed by keys (class template)
<code>unordered_multimap</code> (C++11)	collection of key-value pairs, hashed by keys (class template)

Container adaptors

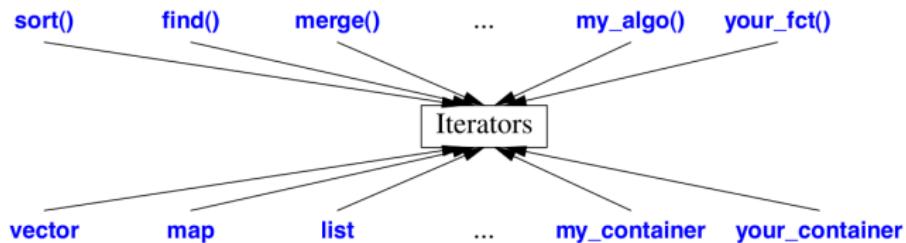
Container adaptors provide a different interface for sequential containers.

<code>stack</code>	adapts a container to provide stack (LIFO data structure) (class template)
<code>queue</code>	adapts a container to provide queue (FIFO data structure) (class template)
<code>priority_queue</code>	adapts a container to provide priority queue (class template)

Iterators

“Iterators are the **glue** that ties standard-library algorithms to their data.

Iterators are the mechanism used to **minimize an algorithm’s dependence** on the data structures on which it operates”



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
- Defined for all using STL containers

Iterators

STL uses iterators to access data in containers

- Access current element with `*iter`
- Accepts `->` alike to pointers
- Move to next element in container `iter++`
- Prefer range-based for loops
- Compare iterators with `==`, `!=`, `<`

Range Access Iterators

- `begin, cbegin` :
returns an iterator to the beginning of a container or array
- `end, cend` :
returns an iterator to the end of a container or array
- `rbegin, crbegin` :
returns a reverse iterator to a container or array
- `rend, crend` :
returns a reverse end iterator for a container or array

Range Access Iterators

Defined for all STL containers:

```
1 #include <array>
2 #include <deque>
3 #include <forward_list>
4 #include <iterator>
5 #include <list>
6 #include <map>
7 #include <regex>
8 #include <set>
9 #include <span>
10 #include <string>
11 #include <string_view>
12 #include <unordered_map>
13 #include <unordered_set>
```

```
1 int main() {
2     vector<double> x{1, 2, 3};
3     for (auto it = x.begin(); it != x.end(); ++it) {
4         cout << *it << endl;
5     }
6     // Map iterators
7     map<int, string> m = {{1, "hello"}, {2, "world"}};
8     map<int, string>::iterator m_it = m.find(1);
9     cout << m_it->first << ":" << m_it->second << endl;
10
11    auto m_it2 = m.find(1); // same thing
12    cout << m_it2->first << ":" << m_it2->second << endl;
13
14    if (m.find(3) == m.end()) {
15        cout << "Key 3 was not found\n";
16    }
17    return 0;
18 }
```

STL Algorithms

- About 80 standard algorithms.
- Defined in `#include <algorithm>`
- They operate on sequences defined by a pair of iterators (for inputs) or a single iterator (for outputs).

Don't reinvent the wheel

- Before writing your own `sort` function :
<http://en.cppreference.com/w/cpp/algorithms>
- When using `std::vector`, `std::array`, etc.
try to avoid writing your own algorithms.
- If you are not using STL containers, then
providing implementations for the standard
iterators will give you access to all the
algorithms for free.
- There is a lot of functions in `std` which are
at least as fast as hand-written ones.

std::sort

```
1 int main() {
2     array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};
3
4     cout << "Before sorting: ";
5     Print(s);
6
7     std::sort(s.begin(), s.end());
8     cout << "After sorting: ";
9     Print(s);
10
11    return 0;
12 }
```

Output:

```
1 Before sorting: 5 7 4 2 8 6 1 9 0 3
2 After sorting: 0 1 2 3 4 5 6 7 8 9
```

std::find

```
1 int main() {
2     const int n1 = 3;
3     std::vector<int> v{0, 1, 2, 3, 4};
4
5     auto result1 = std::find(v.begin(), v.end(), n1);
6
7     if (result1 != std::end(v)) {
8         cout << "v contains: " << n1 << endl;
9     } else {
10        cout << "v does not contain: " << n1 << endl;
11    }
12 }
```

Output:

```
1 v contains: 3
```

std::fill

```
1 int main() {  
2     std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
3  
4     std::fill(v.begin(), v.end(), -1);  
5  
6     Print(v);  
7 }
```

Output:

```
1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

std::count

```
1 int main() {
2     std::vector<int> v{1, 2, 3, 4, 4, 3, 7, 8, 9, 10};
3
4     const int n1 = 3;
5     const int n2 = 5;
6     int num_items1 = std::count(v.begin(), v.end(), n1);
7     int num_items2 = std::count(v.begin(), v.end(), n2);
8     cout << n1 << " count: " << num_items1 << endl;
9     cout << n2 << " count: " << num_items2 << endl;
10
11    return 0;
12 }
```

Output:

```
1 3 count: 2
2 5 count: 0
```

std::count_if

```
1 inline bool div_by_3(int i) { return i % 3 == 0; }
2
3 int main() {
4     std::vector<int> v{1, 2, 3, 3, 4, 3, 7, 8, 9, 10};
5
6     int n3 = std::count_if(v.begin(), v.end(), div_by_3);
7     cout << "# divisible by 3: " << n3 << endl;
8 }
```

Output:

```
1 # divisible by 3: 4
```

std::for_each

```
1 int main() {
2     std::vector<int> nums{3, 4, 2, 8, 15, 267};
3
4     // lambda expression, lecture_9
5     auto print = [] (const int& n) { cout << " " << n; };
6
7     cout << "Numbers:" ;
8     std::for_each(nums.cbegin(), nums.cend(), print);
9     cout << endl;
10
11    return 0;
12 }
```

Output:

```
1 Numbers: 3 4 2 8 15 267
```

std::all_of

```
1 inline bool even(int i) { return i % 2 == 0; }
2 int main() {
3     std::vector<int> v(10, 2);
4     std::partial_sum(v.cbegin(), v.cend(), v.begin());
5     Print(v);
6
7     bool all_even = std::all_of(v.cbegin(), v.cend(), even);
8     if (all_even) {
9         cout << "All numbers are even" << endl;
10    }
11 }
```

Output:

```
1 Among the numbers: 2 4 6 8 10 12 14 16 18 20
2 All numbers are even
```

std::rotate

```
1 int main() {
2     std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
3     cout << "before rotate: ";
4     Print(v);
5
6     std::rotate(v.begin(), v.begin() + 2, v.end());
7     cout << "after rotate: ";
8     Print(v);
9 }
```

Output:

```
1 before rotate: 1 2 3 4 5 6 7 8 9 10
2 after rotate: 3 4 5 6 7 8 9 10 1 2
```

std::transform

```
1 auto Uppercase(char c) { return std::toupper(c); }
2 int main() {
3     const std::string s("hello");
4     std::string S{s};
5     std::transform(s.begin(),
6                   s.end(),
7                   S.begin(),
8                   Uppercase);
9
10    cout << s << endl;
11    cout << S << endl;
12 }
```

Output:

```
1 hello
2 HELLO
```

std::accumulate

```
1 int main() {  
2     std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
3  
4     int sum = std::accumulate(v.begin(), v.end(), 0);  
5  
6     int product = std::accumulate(v.begin(),  
7                                     v.end(),  
8                                     1,  
9                                     std::multiplies());  
10  
11    cout << "Sum      : " << sum << endl;  
12    cout << "Product: " << product << endl;  
13 }
```

Output:

```
1 Sum      : 55  
2 Product: 3628800
```

std::max

```
1 int main() {  
2     using std::max;  
3     cout << "max(1, 9999) : " << max(1, 9999) << endl;  
4     cout << "max('a', 'b') : " << max('a', 'b') << endl;  
5 }
```

Output:

```
1 max(1, 9999) : 9999  
2 max('a', 'b') : b
```

std::min_element

```
1 int main() {  
2     std::vector<int> v{3, 1, 4, 1, 0, 5, 9};  
3  
4     auto result = std::min_element(v.begin(), v.end());  
5     auto min_location = std::distance(v.begin(), result);  
6     cout << "min at: " << min_location << endl;  
7 }
```

Output:

```
1 min at: 4
```

std::minmax_element

```
1 int main() {
2     using std::minmax_element;
3
4     auto v = {3, 9, 1, 4, 2, 5, 9};
5     auto [min, max] = minmax_element(begin(v), end(v));
6
7     cout << "min = " << *min << endl;
8     cout << "max = " << *max << endl;
9 }
```

Output:

```
1 min = 1
2 max = 9
```

std::clamp

```
1 int main() {
2     // value should be between [kMin, kMax]
3     const double kMax = 1.0F;
4     const double kMin = 0.0F;
5
6     cout << std::clamp(0.5, kMin, kMax) << endl;
7     cout << std::clamp(1.1, kMin, kMax) << endl;
8     cout << std::clamp(0.1, kMin, kMax) << endl;
9     cout << std::clamp(-2.1, kMin, kMax) << endl;
10 }
```

Output:

```
1 0.5
2 1
3 0.1
4 0
```

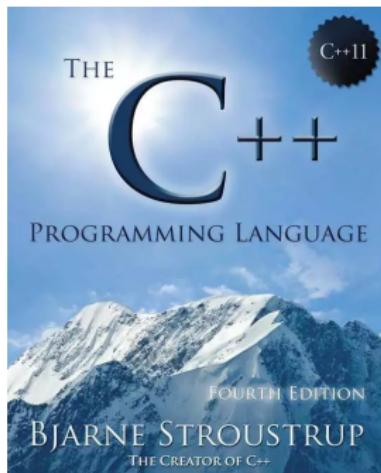
std::sample

```
1 int main() {
2     std::string in = "C++ is cool", out;
3     auto rnd_dev = std::mt19937{random_device{}()};
4     const int kNLetters = 5;
5     std::sample(in.begin(),
6                 in.end(),
7                 std::back_inserter(out),
8                 kNLetters,
9                 rnd_dev);
10
11    cout << "from : " << in << endl;
12    cout << "sample: " << out << endl;
13 }
```

Output:

```
1 from : C++ is cool
2 sample: C++cl
```

References



■ Website:

<http://www.stroustrup.com/4th.html>

References

- **Containers Library**

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

- **Iterators**

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

- **STL Algorithms**

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

Modern C++ for Computer Vision and Image Processing

Lecture 5: I/O Files, Classes

Ignacio Vizzo and 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>

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

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
```

```
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 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 know 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

```
1 #include <filesystem>
2 namespace fs = std::filesystem;
3
4 int main() {
5     fs::create_directories("sandbox/a/b");
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"
```

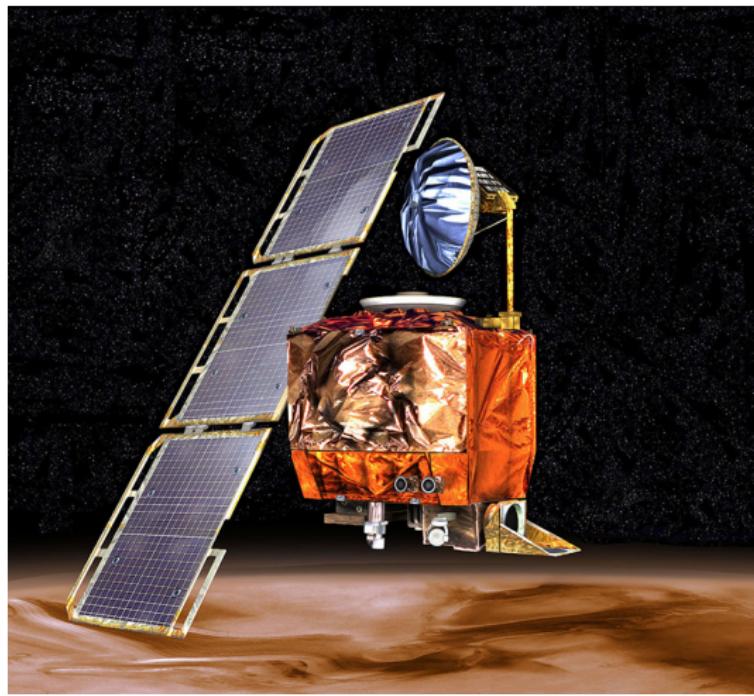
exists

```
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
```

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(milliseconds 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(milliseconds 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 flexibility?

```
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<milliseconds>(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 ???

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)

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.”

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."

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

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`

Example class definition

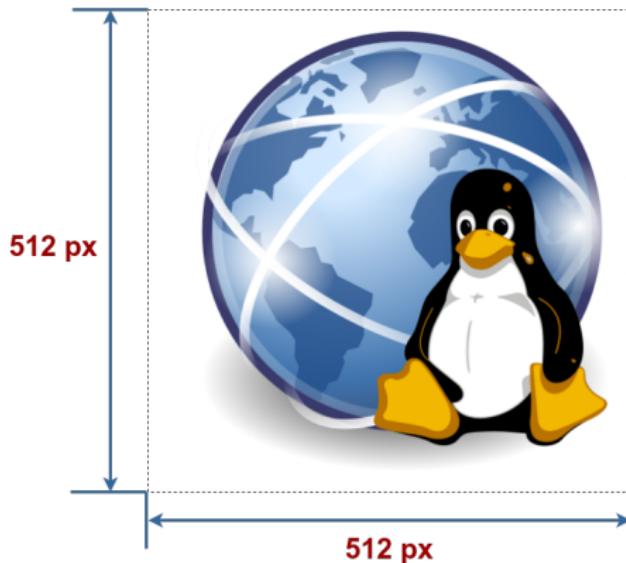
```
1 class Image { // Should be in Image.hpp
2 public:
3     Image(const std::string& file_name);
4     void Draw();
5
6 private:
7     int rows_ = 0; // New in C++11
8     int cols_ = 0; // New in C++11
9 };
10
11 // Implementation omitted here, should be in Image.cpp
12 int main() {
13     Image image("some_image.pgm");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

Classes in our field

```
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 class AxisAlignedBoundingBox : 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

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

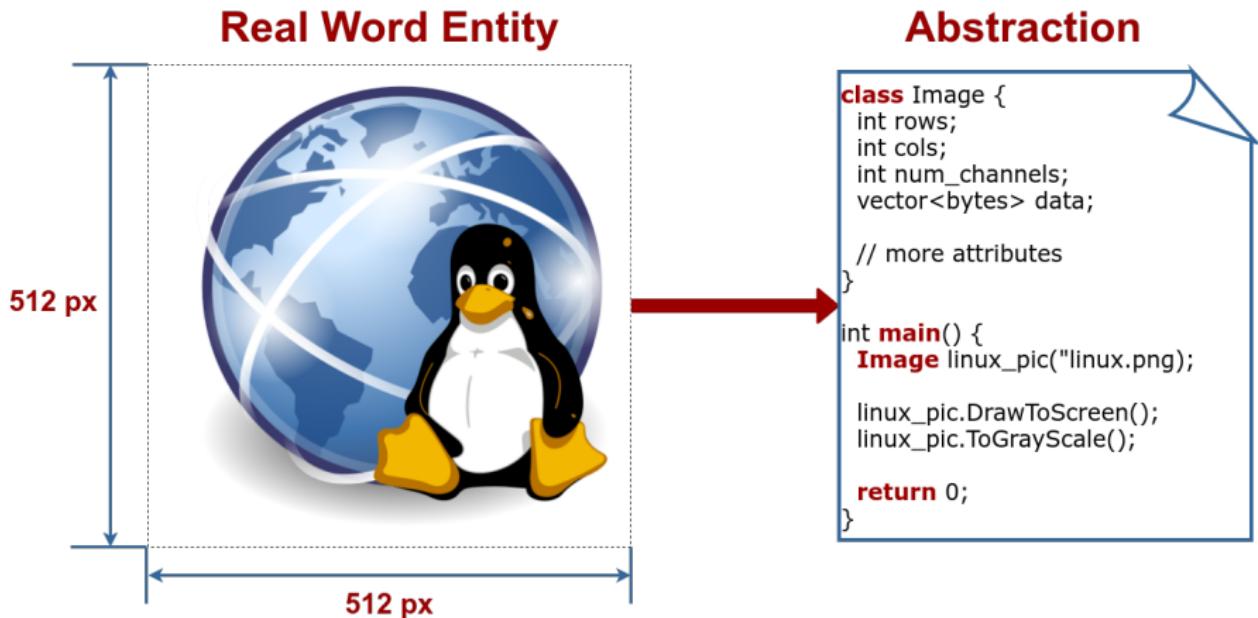
```
1 class Image : public Geometry2D {  
2     public:  
3         /// Width of the image.  
4         int width_ = 0;  
5         /// Height of the image.  
6         int height_ = 0;  
7         /// Number of channels in the image.  
8         int num_of_channels_ = 0;  
9         /// Image storage buffer.  
10        std::vector<uint8_t> data_;  
11    };
```

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 achieved?



Goal achieved?

Open3D::Geometry::Image

```
1 #include <Open3D/Geometry/Image.h>
2
3 using namespace Open3D::Geometry;
4 int main() {
5     Image linux_pic(".data/linux.png");
6
7     auto flipped_linux = linux_pic.FlipHorizontal();
8
9     auto sobel_filter = Image::FilterType::Sobel3Dx;
10    auto filtered_linux = linux_pic.Filter(sobel_filter);
11
12    if (filtered_linux.IsEmpty()) {
13        std::cerr << "Couldn't Filter Image!\n";
14    }
15 }
```

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>

Modern C++ for Computer Vision and Image Processing

Lecture 6: Modern C++ Classes

Ignacio Vizzo and Cyrill Stachniss

Create new types with classes and structs

- Classes are used to **encapsulate data** along with methods to process them
- Every `class` or `struct` defines a new type
- **Terminology:**
 - **Type** or **class** to talk about the defined type
 - A variable of such type is an **instance of class** or an **object**
- Classes allow C++ to be used as an **Object Oriented Programming** language
- `string`, `vector`, etc. are all classes

C++ Class Anatomy

C++ class.cpp class.cpp

```
1  class MyNewType { ← Class Definition
2  public:
3      MyNewType();
4      ~MyNewType(); ← Constructors and Destructors
5
6  public:
7      void MemberFunction1();
8      void MemberFunction2() const; ← Member Functions
9      static void StaticFunction();
10
11 public:
12     MyNewType &operator+=(const MyNewType &other); ← Operators
13     std::ostream &operator<<(std::ostream &os, const MyNewType &obj);
14
15 private:
16     int a_;
17     std::vector<float> data_; ← Data Members
18     MyType2 member_;
19 };
```

Class Glossary

- **Class** Definition.
- **Class** Implementation.
- **Class** data members.
- **Class** Member functions.
- **Class** Constructors.
- **Class** Destructor.
- **Class** setters.
- **Class** getters.
- **Class** operators.
- **Class** static members.
- **Class** Inheritance.

Classes syntax

- Definition starts with the keyword `class`
- Classes have **three access modifiers**: `private`, `protected` and `public`
- By default everything is `private`
- Classes can contain data and functions
- Access members with a `".`
- Have two types of **special functions**:
 - **Constructors**: called upon **creation** of an instance of the class
 - **Destructor**: called upon **destruction** of an instance of the class
- **GOOGLE-STYLE** Use `CamelCase` for class name

What about structs?

- Definition starts with the keyword `struct`:

```
1 struct ExampleStruct {  
2     Type value;  
3     Type value;  
4     Type value;  
5     // No functions!  
6 };
```

- `struct` is a `class` where everything is `public`
- **GOOGLE-STYLE** Use `struct` as a **simple data container**, if it needs a function it should be a `class` instead

Always initialize structs using braced initialization

```
1 #include <iostream>
2 #include <string>
3 struct NamedInt {
4     int num;
5     std::string name;
6 };
7
8 void PrintStruct(const NamedInt& s) {
9     std::cout << s.name << " " << s.num << std::endl;
10 }
11
12 int main() {
13     NamedInt var{1, std::string{"hello"}};
14     PrintStruct(var);
15     PrintStruct({10, std::string{"world"}});
16     return 0;
17 }
```

Data stored in a class

- Classes can store data of any type
- **GOOGLE-STYLE** All data must be `private`
- **GOOGLE-STYLE** Use `snake_case_` with a trailing `_` for `private` data members
- Data should be **set in the Constructor**
- **Cleanup data in the Destructor** if needed

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

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

Constructors and Destructor

- Classes always have **at least one Constructor** and **exactly one Destructor**
- Constructors crash course:
 - Are functions with no **explicit** return type
 - Named exactly as the class
 - There can be many constructors
 - **If there is no explicit constructor an implicit default constructor will be generated**
- Destructor for class `SomeClass`:
 - Is a function named `~SomeClass()`
 - Last function called in the lifetime of an object
 - Generated automatically if not explicitly defined

Many ways to create instances

```
1 class SomeClass {  
2     public:  
3         SomeClass();                      // Default constructor.  
4         SomeClass(int a);                // Custom constructor.  
5         SomeClass(int a, float b);      // Custom constructor.  
6         ~SomeClass();                  // Destructor.  
7     };  
8     // How to use them?  
9     int main() {  
10         SomeClass var_1;                // Default constructor  
11         SomeClass var_2(10);            // Custom constructor  
12         // Type is checked when using {} braces. Use them!  
13         SomeClass var_3{10};            // Custom constructor  
14         SomeClass var_4 = {10};          // Same as var_3  
15         SomeClass var_5{10, 10.0};       // Custom constructor  
16         SomeClass var_6 = {10, 10.0};     // Same as var_5  
17         return 0;  
18     }
```

Setting and getting data

- Use **initializer list** to initialize data
- Name getter functions as the private member they return
- **Avoid setters**, set data in the constructor

```
1 class Student {  
2     public:  
3         Student(int id, string name): id_{id}, name_{name} {}  
4         int id() const { return id_; }  
5         const string& name() const { return name_; }  
6     private:  
7         int id_;  
8         string name_;  
9     };
```

Declaration and definition

- Data members belong to declaration
- Class methods can be defined elsewhere
- Class name becomes part of function name

```
1 // Declare class.  
2 class SomeClass {  
3     public:  
4         SomeClass();  
5         int var() const;  
6     private:  
7         void DoSmth();  
8         int var_ = 0;  
9     };  
10 // Define all methods.  
11 SomeClass::SomeClass() {} // This is a constructor  
12 int SomeClass::var() const { return var_; }  
13 void SomeClass::DoSmth() {}
```

Always initialize members for classes

- C++ 11 allows to initialize variables in-place
- Do not initialize them in the constructor
- No need for an explicit default constructor

```
1 class Student {  
2     public:  
3         // No need for default constructor.  
4         // Getters and functions omitted.  
5     private:  
6         int earned_points_ = 0;  
7         float happiness_ = 1.0f;  
8     };
```

- **Note:** Leave the members of **structs** uninitialized as defining them forbids using brace initialization

Classes as modules

- Prefer encapsulating information that belongs together into a class
- **Separate declaration and definition** of the class into header and source files
- Typically, class `SomeClass` is declared in `some_class.hpp` and is defined in `some_class.cpp`

Const correctness

- `const` after function states that this function **does not change the object**
- Mark all functions that **should not** change the state of the object as `const`
- Ensures that we can pass objects by a `const` reference and still call their functions
- Substantially reduces number of errors

Typical const error



```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4 class Student {
5 public:
6     Student(string name) : name_{name} {}
7     // This function *might* change the object
8     const string& name() { return name_; }
9 private:
10    string name_;
11 };
12 void Print(const Student& student) {
13     cout << "Student: " << student.name() << endl;
14 }
```

```
1 error: passing "const Student" as "this" argument
2     discards qualifiers [-fpermissive]
3     cout << "Student: " << student.name() << endl;
```

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
```

std::move

`std::move` is used to indicate that an object `t` may be “moved from”, i.e. allowing the efficient transfer of resources from `t` to another object.

In particular, `std::move` produces an `xvalue expression` that identifies its argument `t`. It is exactly equivalent to a `static_cast` to an `rvalue` reference type.

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`.

Hands on example

```
1 #include <iostream>
2 #include <string>
3 using namespace std; // Save space on slides.
4 void Print(const string& str) {
5     cout << "lvalue: " << str << endl;
6 }
7 void Print(string&& str) {
8     cout << "rvalue: " << str << endl;
9 }
10 int main() {
11     string hello = "hi";
12     Print(hello);
13     Print("world");
14     Print(std::move(hello));
15     // DO NOT access "hello" after move!
16     return 0;
17 }
```

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;
```



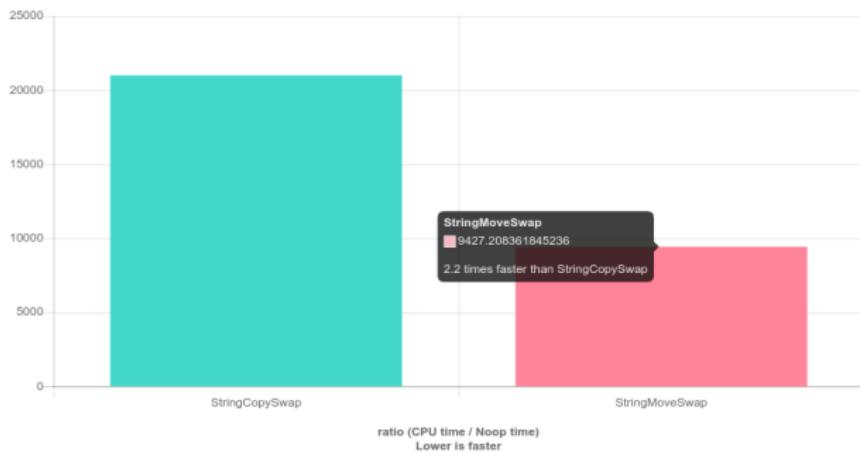
std::move performance

```
1 // MyClass has a private member that contains 200 strings
2 struct MyClass {
3     int id_ = 0;
4     std::vector<std::string> names_{
5         "name", "name", "name", "name", "name", "name", "name", "name",
6         "name", "name", "name", "name", "name", "name", "name", "name",
7         "name", "name", "name", "name", "name", "name", "name", "name",
8         "name", "name", "name", "name", "name", "name", "name", "name",
9         "name", "name", "name", "name", "name", "name", "name", "name",
10        "name", "name", "name", "name", "name", "name", "name", "name",
11        "name", "name", "name", "name", "name", "name", "name", "name",
12        "name", "name", "name", "name", "name", "name", "name", "name",
13        "name", "name", "name", "name", "name", "name", "name", "name",
14        "name", "name", "name", "name", "name", "name", "name", "name",
15        "name", "name", "name", "name", "name", "name", "name", "name",
16        "name", "name", "name", "name", "name", "name", "name", "name",
17        "name", "name", "name", "name", "name", "name", "name", "name",
18        "name", "name", "name", "name", "name", "name", "name", "name",
19        "name", "name", "name", "name", "name", "name", "name", "name",
20        "name", "name", "name", "name", "name", "name", "name", "name",
21        "name", "name", "name", "name", "name", "name", "name", "name",
22        "name", "name", "name", "name", "name", "name", "name", "name",
23        "name", "name", "name", "name", "name", "name", "name", "name",
24        "name", "name", "name", "name", "name", "name", "name", "name",
25        "name", "name", "name", "name", "name", "name", "name", "name",
26        "name", "name", "name", "name", "name", "name", "name", "name"};
27 }
```

std::move performance

```
1 void copy_swap(MyClass& obj1, MyClass& obj2) {
2     MyClass tmp = obj1; // copy obj1 to tmp
3     obj1 = obj2; // copy obj2 to obj1
4     obj2 = tmp; // copy tmp to obj1
5 }
6
7 void move_swap(MyClass& obj1, MyClass& obj2) {
8     MyClass tmp = std::move(obj1); // move obj1 to tmp
9     obj1 = std::move(obj2); // move obj2 to obj1
10    obj2 = std::move(tmp); // move tmp to obj1
11 }
```

std::move performance



Quick Benchmark available to play:
<https://bit.ly/2DFfhko>

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

Custom operators for a class

- Operators are functions with a signature:
`<RETURN_TYPE> operator<NAME>(<PARAMS>)`
- `<NAME>` represents the target operation,
e.g. `>`, `<`, `=`, `==`, `<<` etc.
- Have all attributes of functions
- Always contain word `operator` in name
- All available operators:

<http://en.cppreference.com/w/cpp/language/operators>

Example operator <

```
1 #include <algorithm>
2 #include <vector>
3 class Human {
4 public:
5     Human(int kindness) : kindness_{kindness} {}
6     bool operator<(const Human& other) const {
7         return kindness_ < other.kindness_;
8     }
9
10    private:
11        int kindness_ = 100;
12    };
13 int main() {
14     std::vector<Human> humans = {Human{0}, Human{10}};
15     std::sort(humans.begin(), humans.end());
16     return 0;
17 }
```

Example operator <<

```
1 #include <iostream>
2 #include <vector>
3 class Human {
4 public:
5     int kindness(void) const { return kindness_; }
6 private:
7     int kindness_ = 100;
8 };
9
10 std::ostream& operator<<(std::ostream& os, const Human& human) {
11     os << "This human is this kind: " << human.kindness();
12     return os;
13 }
14
15 int main() {
16     std::vector<Human> humans = {Human{0}, Human{10}};
17     for (auto&& human : humans) {
18         std::cout << human << std::endl;
19     }
20     return 0;
21 }
```

Copy constructor

- **Called automatically** when the object is copied
- For a class MyClass has the signature:
MyClass(const MyClass& other)

```
1 MyClass a;           // Calling default constructor.  
2 MyClass b(a);       // Calling copy constructor.  
3 MyClass c = a;       // Calling copy constructor.
```

Copy assignment operator

- Copy assignment operator is **called automatically** when the object is **assigned a new value** from an **Lvalue**
- For class `MyClass` has a signature:
`MyClass& operator=(const MyClass& other)`
- **Returns a reference** to the changed object
- Use `*this` from within a function of a class to get a reference to the current object

```
1 MyClass a;           // Calling default constructor.  
2 MyClass b(a);       // Calling copy constructor.  
3 MyClass c = a;       // Calling copy constructor.  
4 a = b;               // Calling copy assignment operator.
```

Move constructor

- **Called automatically** when the object is **moved**
- For a class `MyClass` has a signature:
`MyClass(MyClass&& other)`

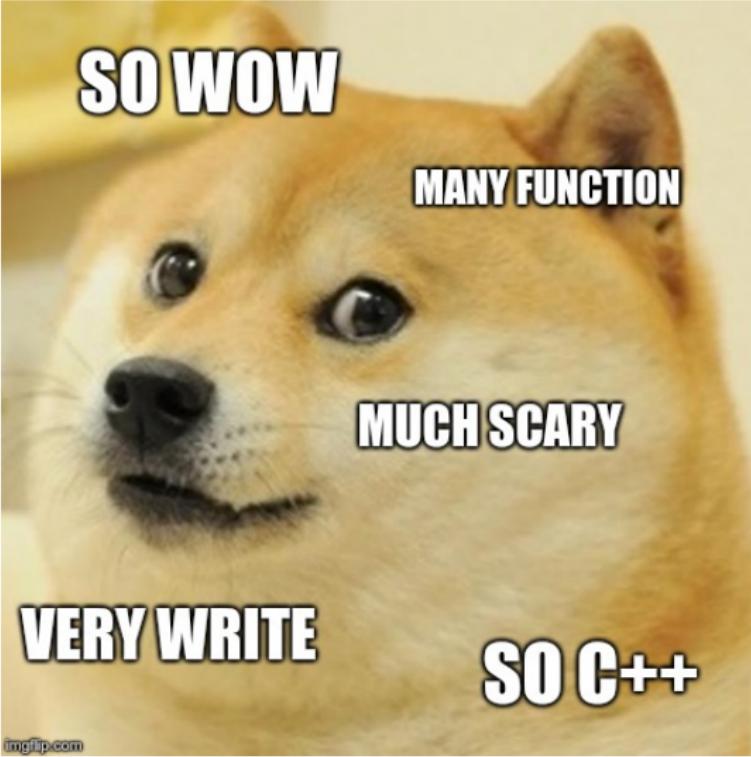
```
1 MyClass a;                                // Default constructors.  
2 MyClass b(std::move(a));      // Move constructor.  
3 MyClass c = std::move(a);    // Move constructor.
```

Move assignment operator

- **Called automatically** when the object is **assigned a new value** from an **Rvalue**
- For class `MyClass` has a signature:
`MyClass& operator=(MyClass&& other)`
- **Returns a reference** to the changed object

```
1 MyClass a;                                // Default constructors.  
2 MyClass b(std::move(a));      // Move constructor.  
3 MyClass c = std::move(a);    // Move constructor.  
4 b = std::move(c);          // Move assignment operator.
```

```
1 class MyClass {
2 public:
3     MyClass() { cout << "default" << endl; }
4     // Copy(&) and Move(&&) constructors
5     MyClass(const MyClass& other) {
6         cout << "copy" << endl;
7     }
8     MyClass(MyClass&& other) {
9         cout << "move" << endl;
10    }
11    // Copy(&) and Move(&&) operators
12    MyClass& operator=(const MyClass& other) {
13        cout << "copy operator" << endl;
14    }
15    MyClass& operator=(MyClass&& other) {
16        cout << "move operator" << endl;
17    }
18 };
19
20 int main() {
21     MyClass a;                                // Calls DEFAULT constructor
22     MyClass b = a;                            // Calls COPY constructor
23     a = b;                                  // Calls COPY assignment operator
24     MyClass c = std::move(a); // Calls MOVE constructor
25     c = std::move(b); // Calls MOVE assignment operator
26 }
```



SO WOW

MANY FUNCTION

MUCH SCARY

VERY WRITE

SO C++

Do I need to define all of them?

- The constructors and operators will be **generated automatically**
- **Under some conditions...**
- Six special functions for class `MyClass`:
 - `MyClass()`
 - `MyClass(const MyClass& other)`
 - `MyClass& operator=(const MyClass& other)`
 - `MyClass(MyClass&& other)`
 - `MyClass& operator=(MyClass&& other)`
 - `~MyClass()`
- **None** of them defined: **all** auto-generated
- **Any** of them defined: **none** auto-generated

Rule of all or nothing

- Try to define **none** of the special functions
- If you **must** define one of them **define all**
- Use **=default** to use default implementation

```
1 class MyClass {  
2     public:  
3         MyClass() = default;  
4         MyClass(MyClass&& var) = default;  
5         MyClass(const MyClass& var) = default;  
6         MyClass& operator=(MyClass&& var) = default;  
7         MyClass& operator=(const MyClass& var) = default;  
8     };
```

Deleted functions

- Any function can be set as `deleted`

```
1 void SomeFunc(...) = delete;
```

- Calling such a function will result in compilation error
- **Example:** remove copy constructors when only one instance of the class must be guaranteed ([Singleton Pattern](#))
- Compiler marks some functions deleted automatically
- **Example:** if a class has a constant data member, the copy/move constructors and assignment operators are implicitly deleted

Static variables and methods

Static member variables of a class

- Exist exactly **once** per class, **not** per object
- The value is equal across all instances
- Must be defined in `*.cpp` files(before `C++17`)

Static member functions of a class

- Do not need to access through an object of the class
- Can access private members but need an object
- **Syntax** for calling:

`ClassName::MethodName(<params>)`

Static variables : “Counted.hpp”

```
1 class Counted {  
2 public:  
3     // Increment the count every time someone creates  
4     // a new object of class Counted  
5     Counted() { Counted::count++; }  
6  
7     // Decrement the count every time someone deletes  
8     // any object of class Counted  
9     ~Counted() { Counted::count--; }  
10  
11    // Static counter member. Keep the count of how  
12    // many objects we've created so far  
13    static int count;  
14};
```

We can access the `count` public member of the `Counted` class through the namespace resolutions operator: “`::`”

Static variables

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4
5 // Include the Counted class declaration and
6 // Initialize the static member of the class only once.
7 // This could be any value
8 #include "Counted.hpp"
9 int Counted::count = 0;
10
11 int main() {
12     Counted a, b;
13     cout << "Count: " << Counted::count << endl;
14     Counted c;
15     cout << "Count: " << Counted::count << endl;
16     return 0;
17 }
```

```
1 #include <cmath>
2
3 class Point {
4 public:
5     Point(int x, int y) : x_(x), y_(y) {}
6
7     static float Dist(const Point& a, const Point& b) {
8         int diff_x = a.x_ - b.x_;
9         int diff_y = a.y_ - b.y_;
10        return sqrt(diff_x * diff_x + diff_y * diff_y);
11    }
12
13    float Dist(const Point& other) {
14        int diff_x = x_ - other.x_;
15        int diff_y = y_ - other.y_;
16        return sqrt(diff_x * diff_x + diff_y * diff_y);
17    }
18
19 private:
20     int x_ = 0;
21     int y_ = 0;
22 };
```

Static member functions

Allow us to define method that does not require an object to call them, but are somehow related to the [Class/Type](#)

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4
5 int main() {
6     Point p1(2, 2);
7     Point p2(1, 1);
8     // Call the static method of the class Point
9     cout << "Dist is " << Point::Dist(p1, p2) << endl;
10
11    // Call the class-method of the Point object p1
12    cout << "Dist is " << p1.Dist(p2) << endl;
13 }
```

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
- http://en.cppreference.com/w/cpp/language/type_alias

Using for type aliasing

```
1 #include <array>
2 #include <memory>
3 template <class T, int SIZE>
4 struct Image {
5     // Can be used in classes.
6     using Ptr = std::unique_ptr<Image<T, SIZE>>;
7     std::array<T, SIZE> data;
8 };
9 // Can be combined with "template".
10 template <int SIZE>
11 using Imagef = Image<float, SIZE>;
12 int main() {
13     // Can be used in a function for type aliasing.
14     using Image3f = Imagef<3>;
15     auto image_ptr = Image3f::Ptr(new Image3f);
16     return 0;
17 }
```

Enumeration classes

- Store an enumeration of options
- Usually derived from `int` type
- Options are assigned consequent numbers
- Mostly used to pick path in `switch`

```
1 enum class EnumType { OPTION_1, OPTION_2, OPTION_3 };
```

- Use values as:
`EnumType::OPTION_1, EnumType::OPTION_2, ...`
- **GOOGLE-STYLE** Name enum type as other types, `CamelCase`
- **GOOGLE-STYLE** Name values as constants `kSomeConstant` or in `ALL_CAPS`

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4 enum class Channel { STDOUT, STDERR };
5 void Print(Channel print_style, const string& msg) {
6     switch (print_style) {
7         case Channel::STDOUT:
8             cout << msg << endl;
9             break;
10        case Channel::STDERR:
11            cerr << msg << endl;
12            break;
13        default:
14            cerr << "Skipping\n";
15    }
16 }
17 int main() {
18     Print(Channel::STDOUT, "hello");
19     Print(Channel::STDERR, "world");
20     return 0;
21 }
```

Explicit values

- By default enum values start from 0
- We can specify custom values if needed
- Usually used with default values

```
1 enum class EnumType {  
2     OPTION_1 = 10,    // Decimal.  
3     OPTION_2 = 0x2,   // Hexadecimal.  
4     OPTION_3 = 13  
5 };
```

Suggested Video

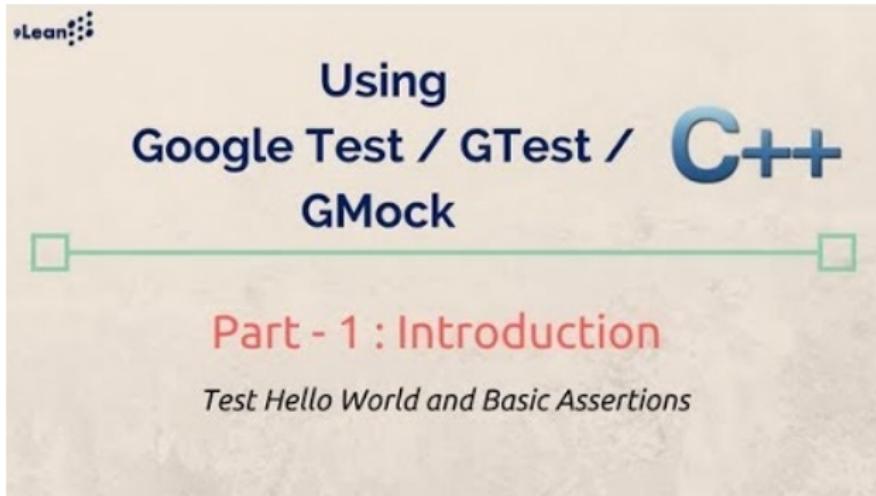
C++ Classes



<https://youtu.be/2BP8NhxjrO0>

Suggested Video

Unit Tests



<https://youtu.be/nbFXI9SDfbk>

References 1

■ **Classes**

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

■ **Data Members**

https://en.cppreference.com/w/cpp/language/data_members

■ **Member Functions**

https://en.cppreference.com/w/cpp/language/member_functions

■ **Static**

<https://en.cppreference.com/w/cpp/language/static>

■ **Operators**

<https://en.cppreference.com/w/cpp/language/operators>

References 2

■ Constructors

<https://en.cppreference.com/w/cpp/language/constructor>

■ Destructor

<https://en.cppreference.com/w/cpp/language/destructor>

■ Copy Constructor

https://en.cppreference.com/w/cpp/language/copy_destructor

■ Move Constructor

https://en.cppreference.com/w/cpp/language/move_constructor

■ Copy Assignment

https://en.cppreference.com/w/cpp/language/copy_assignment

■ Move Assignment

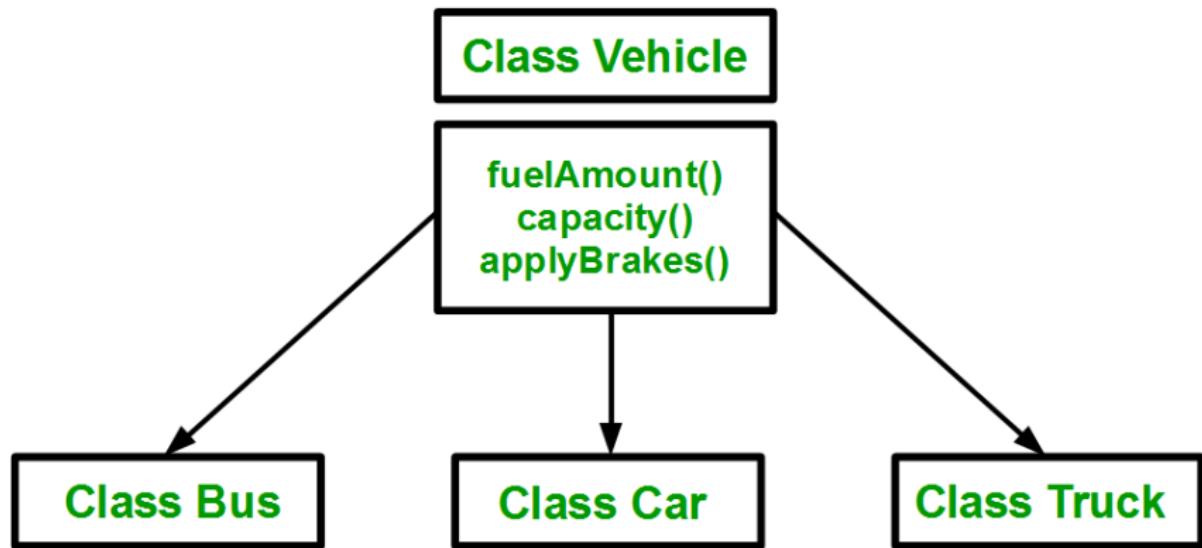
https://en.cppreference.com/w/cpp/language/move_assignment

Modern C++ for Computer Vision and Image Processing

Lecture 7: Object Oriented Design

Ignacio Vizzo and Cyrill Stachniss

Inheritance



C vs C++ Inheritance Example

C Code

```
1 // "Base" class, Vehicle
2 typedef struct vehicle {
3     int seats_;          // number of seats on the vehicle
4     int capacity_;       // amount of fuel of the gas tank
5     char* brand_;        // make of the vehicle
6 } vehicle_t;
```

C++ Code

```
1 class Vehicle {
2     private:
3         int seats_ = 0;          // number of seats on the vehicle
4         int capacity_ = 0;       // amount of fuel of the gas tank
5         string brand_;          // make of the vehicle
```

Inheritance

- Class and struct can **inherit data and functions** from other classes
- There are 3 types of inheritance in C++:
 - `public` [used in this course] GOOGLE-STYLE
 - `protected`
 - `private`
- `public` inheritance keeps all access specifiers of the base class

Public inheritance

- Public inheritance stands for “**is a**” relationship, i.e. if class `Derived` inherits publicly from class `Base` we say, that `Derived` **is a kind of** `Base`

```
1 class Derived : public Base {  
2     // Contents of the derived class.  
3 };
```

- Allows `Derived` to use all `public` and `protected` members of `Base`
- `Derived` still gets its own special functions: constructors, destructor, assignment operators

```
1 #include <iostream>
2 using std::cout; using std::endl;
3 class Rectangle {
4 public:
5     Rectangle(int w, int h) : width_{w}, height_{h} {}
6     int width() const { return width_; }
7     int height() const { return height_; }
8 protected:
9     int width_ = 0;
10    int height_ = 0;
11 };
12 class Square : public Rectangle {
13 public:
14     explicit Square(int size) : Rectangle{size, size} {}
15 };
16 int main() {
17     Square sq(10); // Short name to save space.
18     cout << sq.width() << " " << sq.height() << endl;
19     return 0;
20 }
```

Function overriding

- A function can be declared `virtual`

```
1 virtual Func(<PARAMS>);
```

- If function is `virtual` in `Base` class it can be overridden in `Derived` class:

```
1 Func(<PARAMS>) override;
```

- `Base` can force all `Derived` classes to override a function by making it **pure virtual**

```
1 virtual Func(<PARAMS>) = 0;
```

Overloading vs overriding

- Do not confuse function **overloading** and **overriding**
- **Overloading:**
 - Pick from all functions with the **same name**, but **different parameters**
 - Pick a function **at compile time**
 - Functions don't have to be in a class
- **Overriding:**
 - Pick from functions with the **same arguments and names** in different classes of **one class hierarchy**
 - Pick **at runtime**

Abstract classes and interfaces

- **Abstract class:** class that has at least one pure virtual function
- **Interface:** class that has only pure virtual functions and no data members

How virtual works

- A class with virtual functions has a virtual table
- When calling a function the class checks which of the virtual functions that match the signature should be called
- Called **runtime polymorphism**
- Costs some time but is very convenient

Using interfaces

- Use interfaces when you must **enforce** other classes to implement some functionality
- Allow thinking about classes in terms of **abstract functionality**
- **Hide implementation** from the caller
- Allow to easily extend functionality by simply adding a new class

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 struct Printable { // Saving space. Should be a class.
5     virtual void Print() const = 0;
6 };
7 struct A : public Printable {
8     void Print() const override { cout << "A" << endl; }
9 };
10 struct B : public Printable {
11     void Print() const override { cout << "B" << endl; }
12 };
13 void Print(const Printable& var) { var.Print(); }
14 int main() {
15     Print(A());
16     Print(B());
17     return 0;
18 }
```

Geometry2D and Image

Open3D::Geometry::Geometry2D

```
1 class Geometry2D {  
2     public:  
3         Geometry& Clear() = 0;  
4         bool IsEmpty() const = 0;  
5         virtual Eigen::Vector2d GetMinBound() const = 0;  
6         virtual Eigen::Vector2d GetMaxBound() const = 0;  
7     };
```

Open3D::Geometry::Image

```
1 class Image : public Geometry2D {  
2     public:  
3         Geometry& Clear() override;  
4         bool IsEmpty() const override;  
5         virtual Eigen::Vector2d GetMinBound() const override;  
6         virtual Eigen::Vector2d GetMaxBound() const override;  
7     };
```

Polymorphism

From Greek **polys**, "many, much"
and **morphē**, "form, shape"

-Wiki

- Allows morphing derived classes into their base class type:

```
const Base& base = Derived(...)
```

Polymorphism Example 1

```
1 class Rectangle {
2     public:
3         Rectangle(int w, int h) : width_{w}, height_{h} {}
4         int width() const { return width_; }
5         int height() const { return height_; }
6
7     protected:
8         int width_ = 0;
9         int height_ = 0;
10    };
11
12 class Square : public Rectangle {
13     public:
14         explicit Square(int size) : Rectangle{size, size} {}
15    };

```

Polymorphism Example 1

No real **Polymorphism**, just use all the objects as they are

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     Square sq(10);
6     cout << "Sq:" << sq.width() << " " << sq.height();
7
8     Rectangle rec(10, 15);
9     cout << "Rec:" << sq.width() << " " << sq.height();
10    return 0;
11 }
```

Polymorphism Example 2

```
1 class Rectangle {
2 public:
3     Rectangle(int w, int h) : width_{w}, height_{h} {}
4     int width() const { return width_; }
5     int height() const { return height_; }
6
7     void Print() const {
8         cout << "Rec:" << width_ << " " << height_ << endl;
9     }
}
```

```
1 class Square : public Rectangle {
2 public:
3     explicit Square(int size) : Rectangle{size, size} {}
4     void Print() const {
5         cout << "Sq:" << width_ << " " << height_ << endl;
6     }
7};
```

Polymorphism Example 2

Better than manually calling the getter methods, but still need to explicitly call the `Print()` function for each type of object.
Again, no real **Polymorphism**

```
1 int main() {  
2     Square sq(10);  
3     sq.Print();  
4  
5     Rectangle rec(10, 15);  
6     rec.Print();  
7  
8     return 0;  
9 }
```

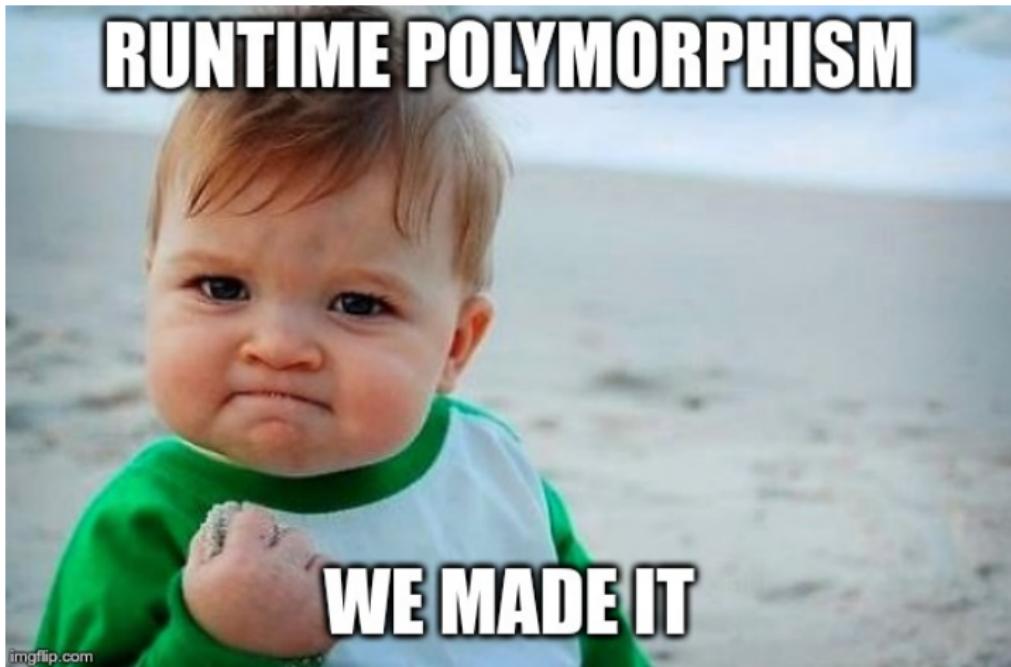
```
1 virtual void Rectangle::Print() const {
2     cout << "Rec:" << width_ << " " << height_ << endl;
3 }
```

```
1 void Square::Print() const override {
2     cout << "Sq:" << width_ << " " << height_ << endl;
3 }
```

```
1 void PrintShape(const Rectangle& rec) { rec.Print(); }
```

```
1 int main() {
2     Square sq(10);
3     Rectangle rec(10, 15);
4
5     PrintShape(rec);
6     PrintShape(sq);
7
8     return 0;
9 }
```

Now we are using **Runtime Polymorphism**, we are printing shapes to the `std::cout` and deciding at runtime with type of `shape` it is



std::vector<Rectangle>

```
1 #include <memory>
2 #include <vector>
3 using std::make_unique;
4 using std::unique_ptr;
5 using std::vector;
6
7 int main() {
8     vector<unique_ptr<Rectangle>> shapes;
9     shapes.emplace_back(make_unique<Rectangle>(10, 15));
10    shapes.emplace_back(make_unique<Square>(10));
11
12    for (const auto &shape : shapes) {
13        shape->Print();
14    }
15
16    return 0;
17 }
```

When is it useful?

- Allows encapsulating the implementation inside a class only asking it to conform to a common interface
- Often used for:
 - Working with all children of some Base class in unified manner
 - Enforcing an interface in multiple classes to force them to implement some functionality
 - In **strategy** pattern, where some complex functionality is outsourced into separate classes and is passed to the object in a modular fashion

Creating a class hierarchy

- Sometimes classes must form a hierarchy
- Distinguish between **is a** and **has a** to test if the classes should be in one hierarchy:
 - Square **is a** Shape: can inherit from Shape
 - Student **is a** Human: can inherit from Human
 - Car **has a** Wheel: should **not** inherit each other
- GOOGLE-STYLE **Prefer composition**, i.e. including an object of another class as a member of your class
- NACHO-STYLE **Don't get too excited**, use it only when improves code performance/readability.

Casting type of variables

- Every variable has a type
- Types can be converted from one to another
- Type conversion is called **type casting**

Casting type of variables

- There are 5 ways of type casting:
 - `static_cast`
 - `reinterpret_cast`
 - `const_cast`
 - `dynamic_cast`
 - C-style cast(unsafe), will try to:
 - `const_cast`
 - `static_cast`
 - `static_cast`, then `const_cast` (change type + remove const)
 - `reinterpret_cast`
 - `reinterpret_cast`, then `const_cast` (change type + remove const)

static_cast

- Syntax: `static_cast<NewType>(variable)`
- Convert type of a variable at compile time
- **Rarely needed to be used explicitly**
- Can happen implicitly for some types,
e.g. `float` can be cast to `int`
- Pointer to an object of a Derived class can
be **upcast** to a pointer of a Base class
- Enum value can be cast to `int` or `float`
- Full specification is complex!

dynamic_cast

- Syntax: `dynamic_cast<Base*>(derived_ptr)`
- Used to convert a pointer to a variable of Derived type to a pointer of a Base type
- Conversion happens at runtime
- If `derived_ptr` cannot be converted to `Base*` returns a `nullptr`
- **GOOGLE-STYLE** Avoid using dynamic casting

reinterpret_cast

- Syntax:

```
reinterpret_cast<NewType>(variable)
```

- Reinterpret the bytes of a variable as another type
- We must know what we are doing!
- Mostly used when writing binary data

const_cast

- Syntax: `const_cast<NewType>(variable)`
- Used to “constify” objects
- Used to “de-constify” objects
- Not widely used

Google Style

- **GOOGLE-STYLE** Do not use C-style casts. Instead, use these C++-style casts when explicit type conversion is necessary.
- **GOOGLE-STYLE** Use brace initialization to convert arithmetic types (e.g. `int64{x}`). This is the safest approach because code will not compile if conversion can result in information loss. The syntax is also concise.

Google Style

- **GOOGLE-STYLE** Use `static_cast` as the equivalent of a C-style cast that does value conversion, when you need to explicitly up-cast a pointer from a class to its superclass, or when you need to explicitly cast a pointer from a superclass to a subclass. In this last case, you must be sure your object is actually an instance of the subclass.

Google Style

- GOOGLE-STYLE Use `const_cast` to remove the `const` qualifier (see `const`).
- GOOGLE-STYLE Use `reinterpret_cast` to do unsafe conversions of pointer types to and from integer and other pointer types. Use this only if you know what you are doing and you understand the aliasing issues.

Using strategy pattern

- If a class relies on complex external functionality use strategy pattern
- Allows to **add/switch functionality** of the class without changing its implementation
- All strategies must conform to one strategy interface

```
1 class Strategy {  
2     public:  
3         virtual void Print() const = 0;  
4 };  
  
1 class StrategyA : public Strategy {  
2     public:  
3         void Print() const override { cout << "A" << endl; }  
4 };  
5  
6 class StrategyB : public Strategy {  
7     public:  
8         void Print() const override { cout << "B" << endl; }  
9 };
```

So far, nothing is new with this source code. We just defined an **interface** and then we derived 2 classes from this **interface** and implemented the virtual methods.

```
1 class MyClass {  
2     public:  
3         explicit MyClass(const Strategy& s) : strategy_(s) {}  
4         void Print() const { strategy_.Print(); }  
5     private:  
6         const Strategy& strategy_;  
7     };  
8 };
```

- `MyClass` holds a `const` reference to an object of type `Strategy`.
- The strategy will be “picked” when we create an object of the class `MyClass`.
- We don’t need to hold a reference to all the `types` of available strategies.
- The `Print` method has nothing to do with the one we’ve defined in `Strategy`.

- Create two different `strategies` objects

```
1  StrategyA strategy_a = StrategyA();  
2  StrategyB strategy_b = StrategyB();
```

- Create 2 objects that will use the `Strategy` pattern. We pick which `Print` strategy to use when we construct these objects.

```
1  MyClass obj_1(strategy_a);  
2  MyClass obj_2(strategy_b);
```

- Use the objects in a “polymorphic” fashion. Both objects will have a `Print` method but they will call different functions according to the `Strategy` we picked when we build the objects.

```
1  obj_1.Print();  
2  obj_2.Print();
```

Do not overuse it

- Only use these patterns when you need to
- If your class should have a single method for some functionality and will never need another implementation don't make it virtual
- Used mostly to **avoid copying code** and to **make classes smaller** by moving some functionality out

Singleton Pattern

- We want only one instance of a given `class`.
- Without C++ this would be a `if/else` mess.
- C++ has a powerfull compiler, we can use it.
- We can make sure that nobody creates more than 1 instance of a given class, **at compile time**.
- Don't over use it, it's easy to learn, but usually hides a **design** error in your code.
- Sometimes is still necessary, and makes your code better.
- You need to use it in homework_7.

Singleton Pattern: How?

- We can **delete** any **class** member functions.
- This also holds true for the special functions:
 - `MyClass()`
 - `MyClass(const MyClass& other)`
 - `MyClass& operator=(const MyClass& other)`
 - `MyClass(MyClass&& other)`
 - `MyClass& operator=(MyClass&& other)`
 - `~MyClass()`
- Any **private** function can only be accessed by member of the class.

Singleton Pattern: How?

- Let's **hide** the default **Constructor** and also the **destructor**.

```
1 class Singleton {  
2     private:  
3         Singleton() = default;  
4         ~Singleton() = default;  
5     };
```

- This completely **disable** the possibility to create a **Singleton** object or destroy it.

Singleton Pattern: How?

- And now let's delete any copy capability:
 - Copy Constructor.
 - Copy Assignment Operator.

```
1 class Singleton {  
2     public:  
3         Singleton(const Singleton&) = delete;  
4         void operator=(const Singleton&) = delete;  
5     };
```

- This completely **disable** the possibility to copy any existing **Singleton** object.

Singleton Pattern: What now?

- Now we need to create at least **one** instance of the **Singleton** class.
- **How?** Compiler to the rescue:
 - We can create **one unique** instance of the class.
 - At compile time ...
 - Using **static** !.

```
1 class Singleton {  
2     public:  
3         static Singleton& GetInstance() {  
4             static Singleton instance;  
5             return instance;  
6         }  
7     };
```

Singleton Pattern: Completed

```
1 class Singleton {  
2     private:  
3         Singleton() = default;  
4         ~Singleton() = default;  
5  
6     public:  
7         Singleton(const Singleton&) = delete;  
8         void operator=(const Singleton&) = delete;  
9         static Singleton& GetInstance() {  
10             static Singleton instance;  
11             return instance;  
12         }  
13     };
```

Singleton Pattern: Usage

```
1 #include "Singleton.hpp"
2
3 int main() {
4     auto& singleton = Singleton::GetInstance();
5     // ...
6     // do stuff with singleton, the only instance.
7     // ...
8
9     Singleton s1;                      // Compiler Error!
10    Singleton s2(singleton);          // Compiler Error!
11    Singleton s3 = singleton;          // Compiler Error!
12
13    return 0;
14 }
```

CRPT Pattern

```
1 #include <boost/core/demangle.hpp>
2 using boost::core::demangle;
3
4 template <typename T>
5 class Printable {
6 public:
7     explicit Printable() {
8         // Always print its type when created
9         cout << demangle(typeid(T).name()) << " created\n";
10    }
11 };
12
13 class Example1 : public Printable<Example1> {};
14 class Example2 : public Printable<Example2> {};
15 class Example3 : public Printable<Example3> {};
```

CRPT Pattern

Usage:

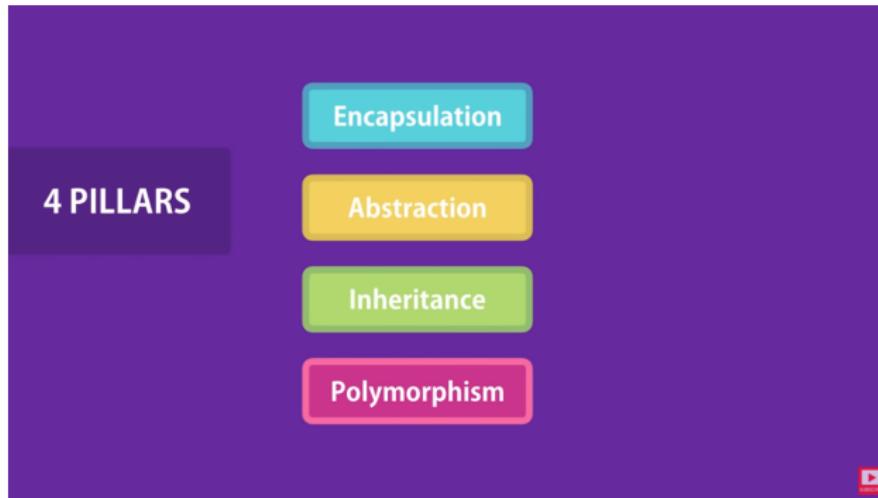
```
1 int main() {  
2     const Example1 obj1;  
3     const Example2 obj2;  
4     const Example3 obj3;  
5     return 0;  
6 }
```

Output:

```
1 Example1 Created  
2 Example2 Created  
3 Example3 Created
```

Suggested Video

Object Oriented Design



<https://youtu.be/pTB0EiLXUC8>

Suggested Video

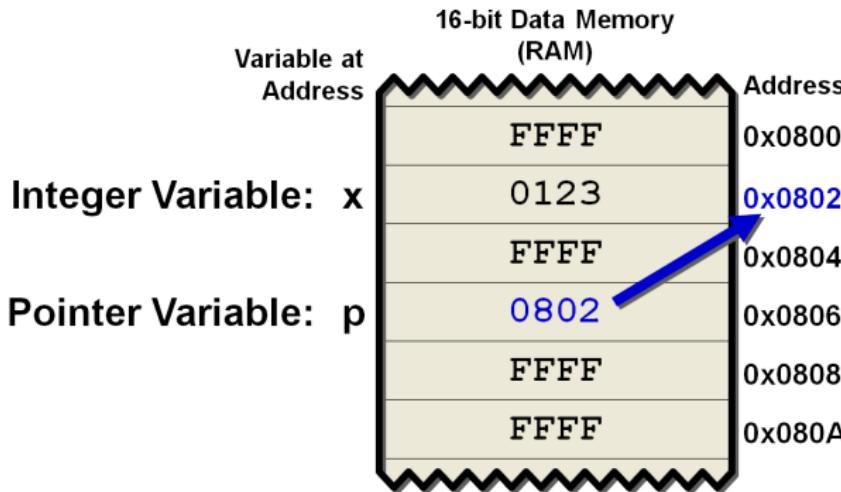
Polymorphism



<https://youtu.be/bP-Trkf8hNA>

Must Watch

Raw Pointers: Skip min 30



<https://www.youtube.com/watch?v=mIrOcFf2crk&t=1729s>

Image Courtesy of Microchip

References

■ Object Oriented Design

- https://en.cppreference.com/w/cpp/language/derived_class
- <https://en.cppreference.com/w/cpp/language/virtual>
- https://en.cppreference.com/w/cpp/language/abstract_class
- <https://en.cppreference.com/w/cpp/language/override>
- <https://en.cppreference.com/w/cpp/language/final>
- <https://en.cppreference.com/w/cpp/language/friend>

■ Type Conversion

- https://en.cppreference.com/w/cpp/language/static_cast
- https://en.cppreference.com/w/cpp/language/dynamic_cast
- https://en.cppreference.com/w/cpp/language/reinterpret_cast
- https://en.cppreference.com/w/cpp/language/const_cast

References: Patterns

■ **Strategy Pattern**

- https://en.wikipedia.org/wiki/Strategy_pattern
- <https://refactoring.guru/design-patterns/strategy/cpp/example>
- <https://stackoverflow.com/a/1008289/11525517>

■ **Singleton Pattern**

- https://en.wikipedia.org/wiki/Singleton_pattern
- <https://refactoring.guru/design-patterns/singleton/cpp/example>

■ **CRPT Pattern**

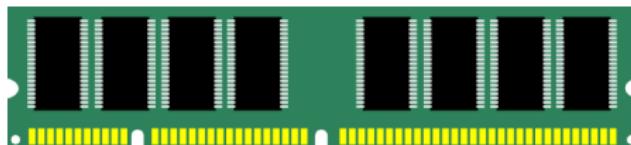
- https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

Modern C++ for Computer Vision and Image Processing

Lecture 8: Memory Management

Ignacio Vizzo and Cyrill Stachniss

Working memory or RAM



<http://www.clipartkid.com>

- Working memory has **linear addressing**
- Every byte has an **address** usually presented in hexadecimal form,
e.g. `0x7fffb7335fdc`
- Any address can be accessed at random
- **Pointer** is a type to store memory addresses

Pointer

- `<TYPE>*` defines a pointer to type `<TYPE>`
- The pointers **have a type**
- Pointer `<TYPE>*` can point **only** to a variable of type `<TYPE>`
- Uninitialized pointers point to a random address
- Always initialize pointers to an address or a `nullptr`

Example:

```
1 int* a = nullptr;
2 double* b = nullptr;
3 YourType* c = nullptr;
```

Non-owning pointers

- Memory pointed to by a raw pointer is not removed when pointer goes out of scope
- Pointers can either own memory or not
- Owning memory means being responsible for its cleanup
- **Raw pointers should never own memory**
- We will talk about **smart pointers** that own memory later

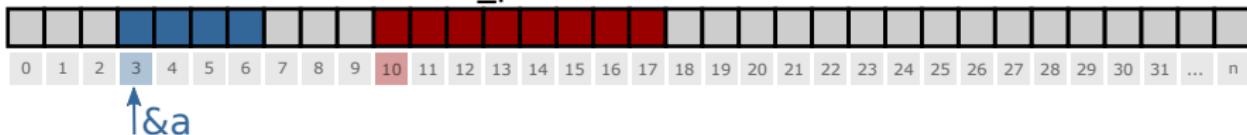
Address operator for pointers

- Operator `&` returns the address of the variable in memory
- Return value type is “pointer to value type”
- `sizeof(pointer)` is 8 bytes in 64bit systems

Example:

```
1 int a = 42;  
2 int* a_ptr = &a;
```

int a; int* a_ptr = &a;

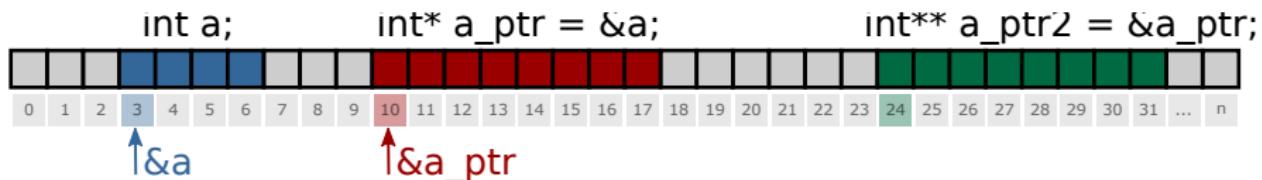


<http://www.cplusplus.com/doc/tutorial/pointers/>

Pointer to pointer

Example:

```
1 int a = 42;  
2 int* a_ptr = &a;  
3 int** a_ptr_ptr = &a_ptr;
```



Pointer dereferencing

- Operator `*` returns the value of the variable to which the pointer points
- Dereferencing of `nullptr`:
Segmentation Fault
- Dereferencing of uninitialized pointer:
Undefined Behavior

Pointer dereferencing

```
1 #include <iostream>
2 using std::cout; using std::endl;
3 int main() {
4     int a = 42;
5     int* a_ptr = &a;
6     int b = *a_ptr;
7     cout << "a = " << a << " b = " << b << endl;
8     *a_ptr = 13;
9     cout << "a = " << a << " b = " << b << endl;
10    return 0;
11 }
```

Output:

```
1 a = 42, b = 42
2 a = 13, b = 42
```



Uninitialized pointer

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int* i_ptr; // BAD! Never leave uninitialized!
6     cout << "ptr address: " << i_ptr << endl;
7     cout << "value under ptr: " << *i_ptr << endl;
8     i_ptr = nullptr;
9     cout << "new ptr address: " << i_ptr << endl;
10    cout << "ptr size: " << sizeof(i_ptr) << " bytes";
11    cout << "(" << sizeof(i_ptr) * 8 << "bit) " << endl;
12    return 0;
13 }
```

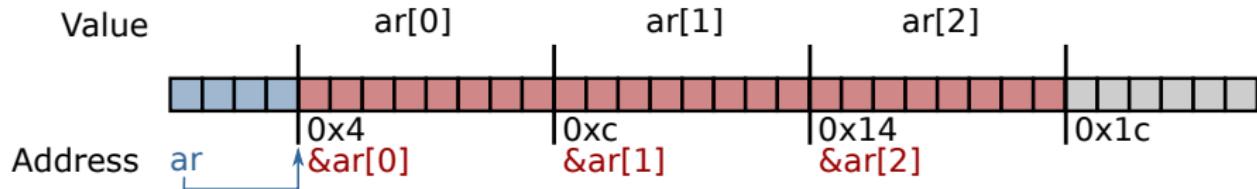
```
1 ptr address: 0x400830
2 value under ptr: -1991643855
3 new ptr address: 0
4 ptr size: 8 bytes (64bit)
```

Important

- Always initialize with a value or a `nullptr`
- Dereferencing a `nullptr` causes a **Segmentation Fault**
- Use `if` to avoid Segmentation Faults

```
1 if(some_ptr) {  
2     // only enters if some_ptr != nullptr  
3 }  
4 if(!some_ptr) {  
5     // only enters if some_ptr == nullptr  
6 }
```

Arrays in memory and pointers



- Array elements are **continuous in memory**
- Name of an array is an alias to a pointer:

```
1 double ar[3];
2 double* ar_ptr = ar;
3 double* ar_ptr = &ar[0];
```

- Get array elements with operator `[]`

Careful! Overflow!

```
1 #include <iostream>
2 int main() {
3     int ar[] = {1, 2, 3};
4     // WARNING! Iterating too far!
5     for (int i = 0; i < 6; i++){
6         std::cout << i << ": value: " << ar[i]
7                         << "\t addr:" << &ar[i] << std::endl;
8     }
9     return 0;
10 }
```

```
1 0: value: 1  addr:0x7ffd17deb4e0
2 1: value: 2  addr:0x7ffd17deb4e4
3 2: value: 3  addr:0x7ffd17deb4e8
4 3: value: 0  addr:0x7ffd17deb4ec
5 4: value: 4196992  addr:0x7ffd17deb4f0
6 5: value: 32764  addr:0x7ffd17deb4f4
```

Using pointers for classes

- Pointers can point to objects of custom classes:

```
1 std::vector<int> vector_int;
2 std::vector<int>* vec_ptr = &vector_int;
3 MyClass obj;
4 MyClass* obj_ptr = &obj;
```

- Call object functions from pointer with `->`

```
1 MyClass obj;
2 obj.MyFunc();
3 MyClass* obj_ptr = &obj;
4 obj_ptr->MyFunc();
```

- `obj->Func() ↔ (*obj).Func()`

Pointers are polymorphic

- Pointers are just like references, but have additional useful properties:
 - Can be reassigned
 - Can point to “nothing” (`nullptr`)
 - Can be stored in a vector or an array

■ Use pointers for polymorphism

```
1 Derived derived;  
2 Base* ptr = &derived;
```

- **Example:** for implementing strategy store a pointer to the strategy interface and initialize it with `nullptr` and check if it is set before calling its methods

```
1 struct AbstractShape {
2     virtual void Print() const = 0;
3 };
4 struct Square : public AbstractShape {
5     void Print() const override { cout << "Square\n"; }
6 };
7 struct Triangle : public AbstractShape {
8     void Print() const override { cout << "Triangle\n"; }
9 };
10
11 int main() {
12     std::vector<AbstractShape*> shapes;
13     Square square;
14     Triangle triangle;
15     shapes.push_back(&square);
16     shapes.push_back(&triangle);
17     for (const auto& shape : shapes) {
18         shape->Print();
19     }
20     return 0;
21 }
```

this pointer

- Every object of a class or a struct holds a pointer to itself
- This pointer is called `this`
- Allows the objects to:
 - Return a reference to themselves: `return *this;`
 - Create copies of themselves within a function
 - Explicitly show that a member belongs to the current object: `this->x();`
 - `this` is a C++ keyword

<https://en.cppreference.com/w/cpp/language/this>

Using `const` with pointers

- Pointers can **point to** a `const` variable:

```
1 // Cannot change value, can reassign pointer.  
2 const MyType* const_var_ptr = &var;  
3 const_var_ptr = &var_other;
```

- Pointers can **be** `const`:

```
1 // Cannot reassign pointer, can change value.  
2 MyType* const var_const_ptr = &var;  
3 var_const_ptr->a = 10;
```

- Pointers can do both at the same time:

```
1 // Cannot change in any way, read-only.  
2 const MyType* const const_var_const_ptr = &var;
```

- Read from right to left to see which `const` refers to what

Memory management structures

Working memory is divided into two parts:

Stack and Heap



stack

<http://www.freestockphotos.biz>



heap

<https://pixabay.com>

Stack memory



- **Static** memory
- Available for **short term** storage (scope)
- **Small / limited** (8 MB Linux typically)
- Memory allocation is **fast**
- **LIFO** (Last in First out) structure
- Items added to top of the stack with **push**
- Items removed from the top with **pop**

Stack memory

stack frame



```
1 #include <stdio.h>
2 int main(int argc, char const* argv[]) {
3     int size = 2;
4     int* ptr = nullptr;
5     {
6         int ar[size];
7         ar[0] = 42;
8         ar[1] = 13;
9         ptr = ar;
10    }
11    for (int i = 0; i < size; ++i) {
12        printf("%d\n", ptr[i]);
13    }
14    return 0;
}
```

command: 2 x pop()

Heap memory



- **Dynamic** memory
- Available for **long** time (program runtime)
- Raw modifications possible with `new` and `delete` (usually encapsulated within a class)
- Allocation is slower than stack allocations



Operators `new` and `new[]`

- User controls memory allocation (unsafe)
- Use `new` to allocate data:

```
1 // pointer variable stored on stack
2 int* int_ptr = nullptr;
3 // 'new' returns a pointer to memory in heap
4 int_ptr = new int;
5
6 // also works for arrays
7 float* float_ptr = nullptr;
8 // 'new' returns a pointer to an array on heap
9 float_ptr = new float[number];
```

- `new` returns an address of the variable on the heap
- **Prefer using smart pointers!**



Operators `delete` and `delete[]`

- **Memory is not freed automatically!**
- User must remember to free the memory
- Use `delete` or `delete[]` to free memory:

```
1 int* int_ptr = nullptr;
2 int_ptr = new int;
3 // delete frees memory to which the pointer points
4 delete int_ptr;
5
6 // also works for arrays
7 float* float_ptr = nullptr;
8 float_ptr = new float[number];
9 // make sure to use 'delete[]' for arrays
10 delete[] float_ptr;
```

- **Prefer using smart pointers!**

Example: heap memory

```
1 #include <iostream>
2 using std::cout; using std::endl;
3 int main() {
4     int size = 2; int* ptr = nullptr;
5     {
6         ptr = new int[size];
7         ptr[0] = 42; ptr[1] = 13;
8     } // End of scope does not free heap memory!
9     // Correct access, variables still in memory.
10    for (int i = 0; i < size; ++i) {
11        cout << ptr[i] << endl;
12    }
13    delete[] ptr; // Free memory.
14    for (int i = 0; i < size; ++i) {
15        // Accessing freed memory. UNDEFINED!
16        cout << ptr[i] << endl;
17    }
18    return 0;
19 }
```

Memory leak

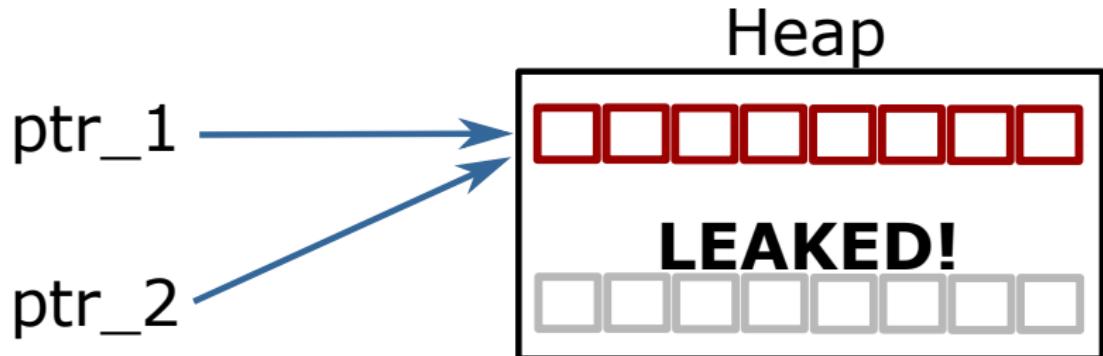
- Can happen when working with Heap memory if we are not careful
- **Memory leak**: memory allocated on Heap access to which has been lost

Memory leak

- Can happen when working with Heap memory if we are not careful
- **Memory leak**: memory allocated on Heap access to which has been lost

Memory leak

- Can happen when working with Heap memory if we are not careful
- **Memory leak**: memory allocated on Heap access to which has been lost



Memory leak (delete)

```
1 int main() {
2     int *ptr_1 = nullptr;
3     int *ptr_2 = nullptr;
4
5     // Allocate memory for two bytes on the heap.
6     ptr_1 = new int;
7     ptr_2 = new int;
8     cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
9
10    // Overwrite ptr_2 and make it point where ptr_1
11    ptr_2 = ptr_1;
12
13    // ptr_2 overwritten, no chance to access the memory.
14    cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
15    delete ptr_1;
16    delete ptr_2;
17    return 0;
18 }
```

Error: double free or corruption

```
1 ptr_1: 0x10a3010, ptr_2: 0x10a3070
2 ptr_1: 0x10a3010, ptr_2: 0x10a3010
3 *** Error: double free or corruption (fasttop): 0
   x00000000010a3010 ***
```

- The memory under address `0x10a3070` is **never** freed
- Instead we try to free memory under `0x10a3010` **twice**
- Freeing memory twice is an error

Tools to the rescue

- Standard tools like: `valgrind ./my_program`
- Compiler flags `-fsanitize=address`
- Stackoverflow!



code-fsanitize=address

```
1 =====
2 ==19747==ERROR: AddressSanitizer: attempting double-
3     free on 0x602000000010 in thread T0:
4 # ... more stuff
5 0x602000000010 is located 0 bytes inside of 4-byte
6 # ... even more stuff
7 SUMMARY: AddressSanitizer: double-free in operator
8     delete(void*, unsigned long)
9 # ... even more more stuff
10 ==19747==ABORTING
```

valgrind output

```
1 HEAP SUMMARY:
2     in use at exit: 4 bytes in 1 blocks
3 total heap usage: 4 allocs, 4 frees, 76,808 bytes
4     allocated
5
5 LEAK SUMMARY:
6     definitely lost: 4 bytes in 1 blocks
7     indirectly lost: 0 bytes in 0 blocks
8     possibly lost: 0 bytes in 0 blocks
9     still reachable: 0 bytes in 0 blocks
10    suppressed: 0 bytes in 0 blocks
11 Rerun with --leak-check=full to see details of leaked
12    memory
13 For counts of detected and suppressed errors, rerun
14    with: -v
15
16 ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0
17    from 0)
```

Memory leak example

```
1 int main() {
2     double *data = nullptr;
3     size_t size = pow(1024, 3) / 8;    // Produce 1GB
4
5     for (int i = 0; i < 4; ++i) {
6         // Allocate memory for the data.
7         data = new double[size];
8         std::fill(data, data + size, 1.23);
9         // Do some important work with the data here.
10        cout << "Iteration: " << i << " done. " << (i + 1)
11            << " GiB has been allocated!" << endl;
12    }
13
14    // This will only free the last allocation!
15    delete[] data;
16    int unused;
17    std::cin >> unused;    // Wait for user.
18    return 0;
19 }
```

Memory leak example

- If we run out of memory an `std::bad_alloc` error is thrown
- Be careful running this example, everything might become slow

```
1 # ...
2 Iteration: 19 done. 20 GiB has been allocated!
3 Iteration: 20 done. 21 GiB has been allocated!
4 Iteration: 21 done. 22 GiB has been allocated!
5 Iteration: 22 done. 23 GiB has been allocated!
6 terminate called after throwing an instance of 'std::
   bad_alloc'
7   what():  std::bad_alloc
8 [1] 30561 abort (core dumped) ./memory_leak_2
```

Dangling pointer

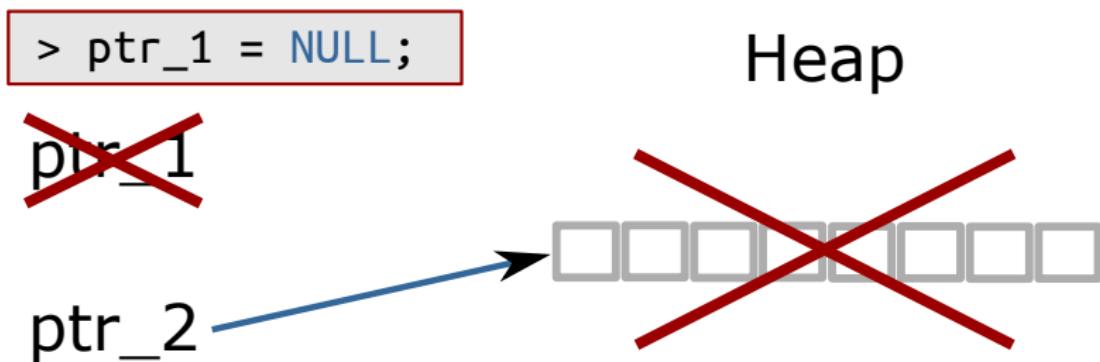
```
1 int* ptr_1 = some_heap_address;
2 int* ptr_2 = some_heap_address;
3 delete ptr_1;
4 ptr_1 = nullptr;
5 // Cannot use ptr_2 anymore! Behavior undefined!
```

Dangling pointer

```
1 int* ptr_1 = some_heap_address;
2 int* ptr_2 = some_heap_address;
3 delete ptr_1;
4 ptr_1 = nullptr;
5 // Cannot use ptr_2 anymore! Behavior undefined!
```

Dangling pointer

```
1 int* ptr_1 = some_heap_address;
2 int* ptr_2 = some_heap_address;
3 delete ptr_1;
4 ptr_1 = nullptr;
5 // Cannot use ptr_2 anymore! Behavior undefined!
```



Dangling pointer

- **Dangling Pointer**: pointer to a freed memory
- Think of it as the opposite of a memory leak
- Dereferencing a dangling pointer causes **undefined behavior**



Dangling pointer example

```
1 #include <iostream>
2 using std::cout; using std::endl;
3 int main() {
4     int size = 5;
5     int *ptr_1 = new int[size];
6     int *ptr_2 = ptr_1;    // Point to same data!
7     ptr_1[0] = 100;        // Set some data.
8     cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
9     cout << "ptr_2[0]: " << ptr_2[0] << endl;
10    delete[] ptr_1;    // Free memory.
11    ptr_1 = nullptr;
12    cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
13    // Data under ptr_2 does not exist anymore!
14    cout << "ptr_2[0]: " << ptr_2[0] << endl;
15    return 0;
16 }
```

Even worse when used in functions



```
1 #include <stdio.h>
2 // data processing
3 int* GenerateData(int size);
4 void UseDataForGood(const int* const data, int size);
5 void UseDataForBad(const int* const data, int size);
6 int main() {
7     int size = 10;
8     int* data = GenerateData(size);
9     UseDataForGood(data, size);
10    UseDataForBad(data, size);
11    // Is data pointer valid here? Should we free it?
12    // Should we use 'delete[]' or 'delete'?
13    delete[] data; // ??????????????
14    return 0;
15 }
```



Memory leak or dangling pointer

```
1 void UseDataForGood(const int* const data, int size) {  
2     // Process data, do not free. Leave it to caller.  
3 }  
4 void UseDataForBad(const int* const data, int size) {  
5     delete[] data;    // Free memory!  
6     data = nullptr;  // Another problem - this does  
7     nothing!  
 }
```

- **Memory leak** if nobody has freed the memory
- **Dangling Pointer** if somebody has freed the memory in a function

RAII

- Resource Allocation Is Initialization.
- New object → allocate memory
- Remove object → free memory
- Objects **own** their data!

```
1 class MyClass {  
2     public:  
3         MyClass() { data_ = new SomeOtherClass; }  
4         ~MyClass() {  
5             delete data_;  
6             data_ = nullptr;  
7         }  
8     private:  
9         SomeOtherClass* data_;  
10    };
```

- Still cannot copy an object of **MyClass!!!**



```
1 struct SomeOtherClass {};
2 class MyClass {
3     public:
4         MyClass() { data_ = new SomeOtherClass; }
5         ~MyClass() {
6             delete data_;
7             data_ = nullptr;
8         }
9     private:
10    SomeOtherClass* data_;
11 };
12 int main() {
13     MyClass a;
14     MyClass b(a);
15     return 0;
16 }
```

```
1 *** Error in `raii_example':
2 double free or corruption: 0x000000000877c20 ***
```

Shallow vs deep copy

- **Shallow copy:** just copy pointers, not data
- **Deep copy:** copy data, create new pointers
- Default copy constructor and assignment operator implement shallow copying
- RAII + shallow copy → **dangling pointer**
- RAII + Rule of All Or Nothing → **correct**
- **Use smart pointers instead!**

Smart pointers



Raw pointers are hard to love

1. Its declaration doesn't indicate whether it points to a single `object` or to an `array`.
2. Its declaration reveals **nothing** about whether you should destroy what it points to when you're done using it, i.e., if the pointer **owns** the thing it points to.
3. If you determine that you should **destroy** what the pointer points to, there's no way to tell how. Should you use `delete`, or is there a different `destruction` mechanism (e.g., a dedicated destruction function the pointer should be passed to)?

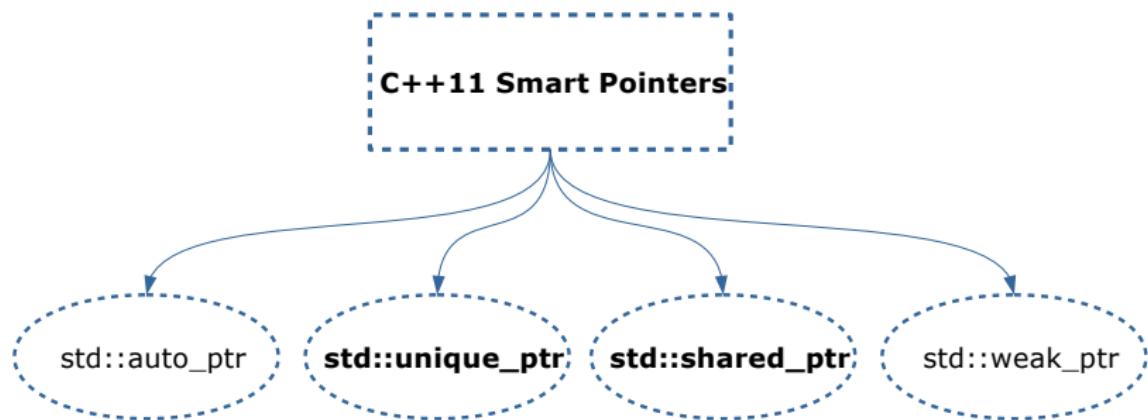
Raw pointers are hard to love

4. If you manage to find out that `delete` is the way to go, Reason 1 means it may not be possible to know whether to use the single-object form ("`delete`") or the `array` form ("`delete []`"). If you use the wrong form, results are **undefined**.
5. There's typically no way to tell if the pointer **dangles**, i.e., points to memory that no longer holds the object the pointer is supposed to point to. Dangling pointers arise when objects are destroyed while pointers still point to them.

Smart pointers

- Smart pointers wrap a raw pointer into a class and manage its lifetime (**RAII**)
- Smart pointers are **all about ownership**
- Always use smart pointers when the pointer should **own heap memory**
- **Only use them with heap memory!**
- Still use raw pointers for non-owning pointers and simple address storing
- `#include <memory>` to use smart pointers

C++11 smart pointers **types**



We will focus on 2 types of smart pointers:

- `std::unique_ptr`
- `std::shared_ptr`

Smart pointers manage memory!

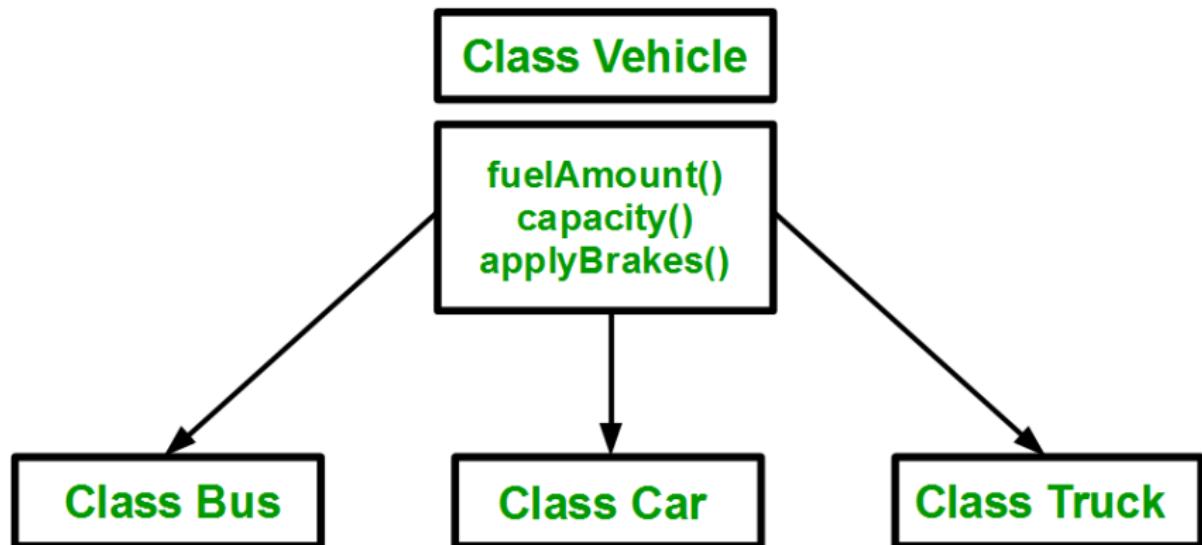
Smart pointers apart from memory allocation behave exactly as raw pointers:

- Can be set to `nullptr`
- Use `*ptr` to dereference `ptr`
- Use `ptr->` to access methods
- Smart pointers are polymorphic

Additional functions of smart pointers:

- `ptr.get()` returns a raw pointer that the smart pointer manages
- `ptr.reset(raw_ptr)` stops using currently managed pointer, freeing its memory if needed, sets `ptr` to `raw_ptr`

std::unique_ptr **example**



std::unique_ptr example

- Create an `unique_ptr` to a type `Vehicle`

```
1 std::unique_ptr<Vehicle> vehicle_1 =  
2 std::make_unique<Bus>(20, 10, "Volkswagen", "LPM_");  
3  
4 std::unique_ptr<Vehicle> vehicle_2 =  
5 std::make_unique<Car>(4, 60, "Ford", "Sony");
```

- Now you can have fun as we had with `raw pointers`

```
1 // vehicle_x is a pointer, so we can use it as it is  
2 vehicle_1->Print();  
3 vehicle_2->Print();
```

std::unique_ptr example

- `unique_ptr` are **unique**: This means that we can move stuff but **not** copy:

```
1 vehicle_2 = std::move(vehicle_1);
```

- Address of the pointers **before** the move:

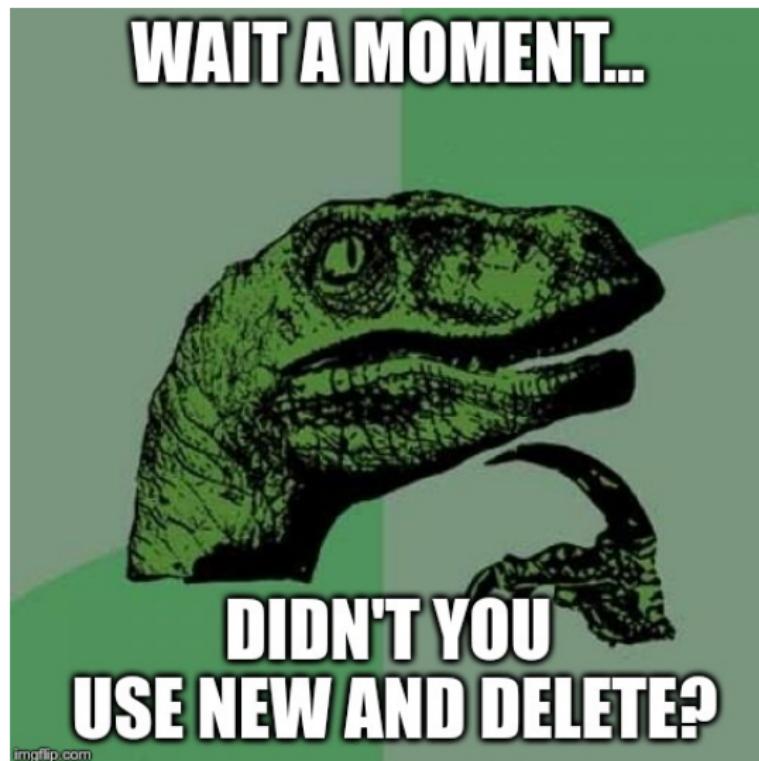
```
1 cout << "vehicle_1 = " << vehicle_1.get() << endl;
2 cout << "vehicle_2 = " << vehicle_2.get() << endl;
```

```
1 vehicle_1 = 0x56330247ce70
2 vehicle_2 = 0x56330247cec0
```

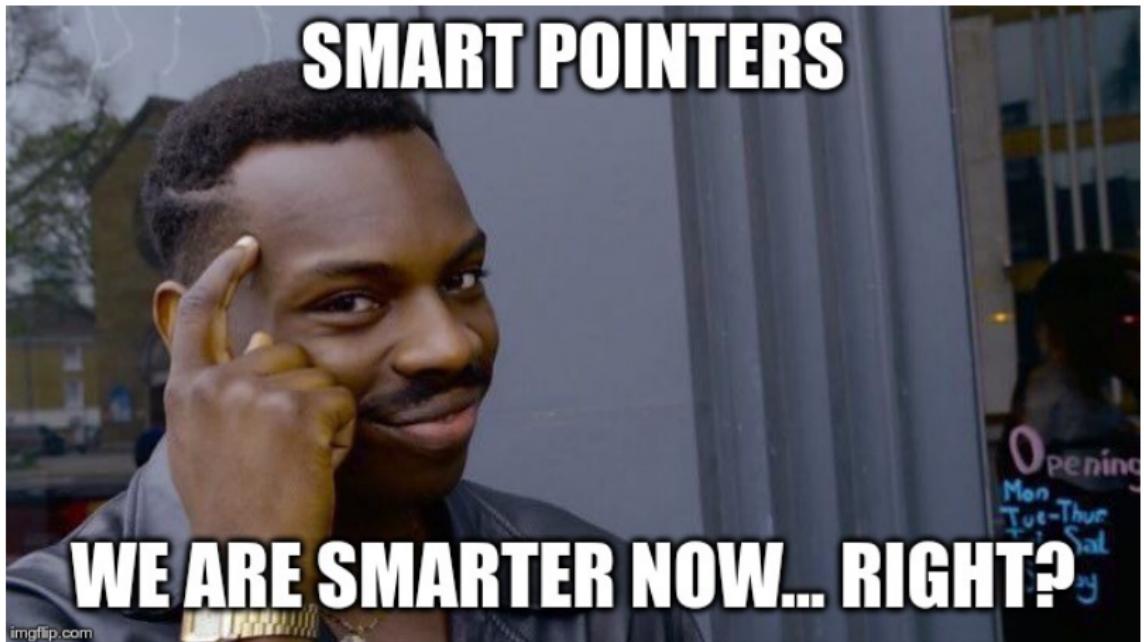
- Address of the pointers **after** the move:

```
1 vehicle_2 = 0x56330247ce70
2 vehicle_1 = 0
```

std::unique_ptr **example**



std::unique_ptr **example**



Unique pointer (std::unique_ptr)

- Constructor of a unique pointer takes **ownership** of a provided raw pointer
- **No runtime overhead** over a raw pointer
- Syntax for a unique pointer to type **Type**:

```
1 #include <memory>
2 // Using default constructor Type();
3 auto p = std::unique_ptr<Type>(new Type);
4 // Using constructor Type(<params>);
5 auto p = std::unique_ptr<Type>(new Type(<params>));
```

- From C++14 on:

```
1 // Forwards <params> to constructor of unique_ptr
2 auto p = std::make_unique<Type>(<params>);
```

What makes it “unique”

- Unique pointer **has no copy constructor**
- Cannot be copied, **can be moved**
- Guarantees that memory is **always** owned by a single `std::unique_ptr`
- A non-null `std::unique_ptr` always owns what it points to.
- Moving a `std::unique_ptr` transfers ownership from the source pointer to the destination pointer. (The source pointer is set to `nullptr`.)

Shared pointer (std::shared_ptr)

- What if we want to use the same `pointer` for different resources?
- An object accessed via `std::shared_ptr`s has its lifetime managed by those pointers through **shared** ownership.
- No specific `std::shared_ptr` owns the object.
- When the last `std::shared_ptr` pointing to an object stops pointing there, that `std::shared_ptr` destroys the object it points to.

Shared pointer (std::shared_ptr)

- Constructed just like a `unique_ptr`
- Can be copied
- Stores a usage counter and a raw pointer
 - Increases usage counter when copied
 - Decreases usage counter when destructed
- Frees memory when counter reaches 0
- Can be initialized from a `unique_ptr`
- Syntax:

```
1 #include <memory>
2 // Using default constructor Type();
3 auto p = std::shared_ptr<Type>(new Type);
4 auto p = std::make_shared<Type>();
5
6 // Using constructor Type(<params>);
7 auto p = std::shared_ptr<Type>(new Type(<params>));
8 auto p = std::make_shared<Type>(<params>);
```

Shared pointer

```
1 class MyClass {
2 public:
3     MyClass() { cout << "I'm alive!\n"; }
4     ~MyClass() { cout << "I'm dead... :(\n"; }
5 };
6
7 int main() {
8     auto a_ptr = std::make_shared<MyClass>();
9     cout << a_ptr.use_count() << endl;
10    {
11        auto b_ptr = a_ptr;
12        cout << a_ptr.use_count() << endl;
13    }
14    cout << "Back to main scope\n";
15    cout << a_ptr.use_count() << endl;
16    return 0;
17 }
```

When to use what?

- Use smart pointers when the pointer **must manage memory**
- By default use `unique_ptr`
- If multiple objects must **share** ownership over something, use a `shared_ptr` to it
- Think of any free standing `new` or `delete` as of a memory leak or a dangling pointer:
 - Don't use `delete`
 - Allocate memory with `make_unique`, `make_shared`
 - Only use `new` in smart pointer constructor if cannot use the functions above

Typical beginner error

```
1 int main() {
2     // Allocate a variable in the stack
3     int a = 42;
4
5     // Create a pointer to that part of the memory
6     int* ptr_to_a = &a;
7
8     // Know stuff about pointers eh?
9     auto a_unique_ptr = std::unique_ptr<int>(ptr_to_a);
10
11    // Same happens with std::shared_ptr.
12    auto a_shared_ptr = std::shared_ptr<int>(ptr_to_a);
13
14    std::cout << "Program terminated correctly!!!\n";
15    return 0;
16 }
```

Typical beginner error

```
1 int* ptr_to_a = &a;  
2  
3 // Know stuff about pointers eh?  
4 auto a_unique_ptr = std::unique_ptr<int>(ptr_to_a);  
5  
6 // Same happens with std::shared_ptr.  
7 auto a_shared_ptr = std::shared_ptr<int>(ptr_to_a);
```

```
1 Program terminated correctly!!!  
2 munmap_chunk(): invalid pointer  
3 [1]    4455 abort (core dumped)  ./wrong_unique
```

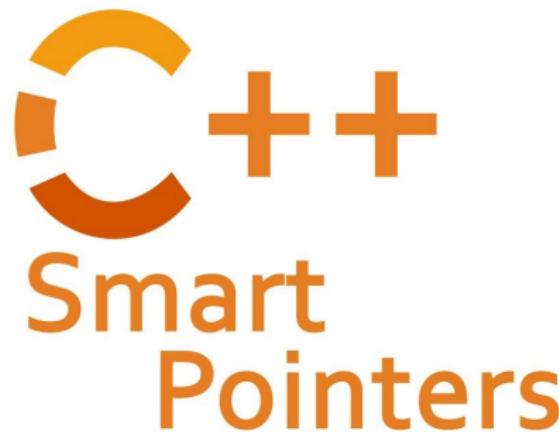
- Create a **smart pointer** from a **pointer** to a stack-managed variable
- The variable ends up being owned both by the **smart pointer** and the stack and gets deleted twice → **Error!**

Polymorphism example using smart pointers

```
1 #include <memory>
2 #include <vector>
3 using std::make_unique;
4 using std::unique_ptr;
5 using std::vector;
6
7 int main() {
8     vector<unique_ptr<Rectangle>> shapes;
9     shapes.emplace_back(make_unique<Rectangle>(10, 15));
10    shapes.emplace_back(make_unique<Square>(10));
11
12    for (const auto &shape : shapes) {
13        shape->Print();
14    }
15
16    return 0;
17 }
```

Suggested Video

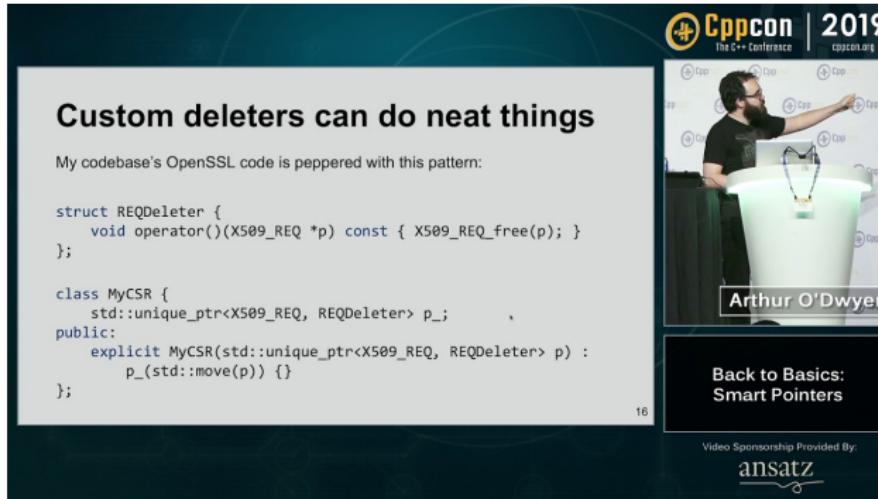
Smart Pointers (short)



<https://youtu.be/UOB7-B2MfwA>

Suggested Video

Smart Pointers (in deep)



Custom deleters can do neat things

My codebase's OpenSSL code is peppered with this pattern:

```
struct REQDeleter {
    void operator()(X509_REQ *p) const { X509_REQ_free(p); }
};

class MyCSR {
    std::unique_ptr<X509_REQ, REQDeleter> p_;
public:
    explicit MyCSR(std::unique_ptr<X509_REQ, REQDeleter> p) :
        p_(std::move(p)) {}
};
```

16

Arthur O'Dwyer

Back to Basics:
Smart Pointers

Video Sponsorship Provided By:
ansatz

<https://youtu.be/xGDLkt-jBJ4>

References

- **Dynamic Memory Management**

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

- **Intro to Smart Pointers**

https://en.cppreference.com/book/intro/smart_pointers

- **Shared Pointers**

https://en.cppreference.com/w/cpp/memory/shared_ptr

- **Unique Pointers**

https://en.cppreference.com/w/cpp/memory/unique_ptr

Modern C++ for Computer Vision and Image Processing

Lecture 09: Templates

Ignacio Vizzo and Cyrill Stachniss

Generic programming

What is Programming?

- “The craft of writing useful, maintainable, and extensible source code which can be interpreted or compiled by a computing system to perform a meaningful task.”
—Wikibooks

What is Meta-Programming?

- “The writing of computer programs that manipulate other programs (or themselves) as if they were data.” —Anders Hejlsberg

Meaning of template

Dictionary Definitions:

- Something that serves as a model for others to copy
- A preset format for a document or file
- Something that is used as a pattern for producing other similar things

Meaning of template

C++ Definitions:

A template is a C++ entity that defines one of the following:

- A family of classes (class template), which may be nested classes.
- A family of functions (function template), which may be member functions.

Motivation: Generic functions

abs():

```
1 double abs(double x) { return (x >= 0) ? x : -x; }  
2 int abs(int x) { return (x >= 0) ? x : -x; }
```

And then also for:

- long
- int
- float
- complex types?
- Maybe char types?
- Maybe short?
- Where does this end?

Motivation: Generic functions

C-style, C99 Standard:

- `abs (int)`
- `labs (long)`
- `llabs (long long)`
- `imaxabs (intmax_t)`
- `fabsf (float)`
- `fabs (double)`
- `fabsl (long double)`
- `cabsf (_Complex float)`
- `cabs (_Complex double)`
- `cabsl (_Complex long double)`

Function Templates

abs<T>():

```
1 template <typename T>
2 T abs(T x) {
3     return (x >= 0) ? x : -x;
4 }
```

- Function templates are not functions.
 - **They are templates for making functions**
- Don't pay for what you don't use:
 - **If nobody calls abs<int>, it won't be instantiated by the compiler at all.**

Template functions

- Use keyword `template`

```
1 template <typename T, typename S>
2 T awesome_function(const T& var_t, const S& var_s) {
3     // some dummy implementation
4     T result = var_t;
5     return result;
6 }
```

- `T` and `S` can be any type.
- A **function template** defines a **family** of functions.

Using Function Templates

```
1 template <typename T>
2 T abs(T x) {
3     return (x >= 0) ? x : -x;
4 }
5
6 int main() {
7     const double x = 5.5;
8     const int y = -5;
9
10    auto abs_x = abs<double>(x);
11    int abs_y = abs<int>(y);
12
13    double abs_x_2 = abs(x); // type-deduction
14    auto abs_y_2 = abs(y); // type-deduction
15 }
```

Template classes

```
1 template <class T>
2 class MyClass {
3 public:
4     MyClass(T x) : x_(x) {}
5
6 private:
7     T x_;
8 };
```

- Classes templates are not classes.
 - **They are templates for making classes**
- Don't pay for what you don't use:
 - **If nobody calls MyClass<int>, it won't be instantiated by the compiler at all.**

Template classes usage

```
1 template <class T>
2 class MyClass {
3 public:
4     MyClass(T x) : x_(x) {}
5
6 private:
7     T x_;
8 };
9
10 int main() {
11     MyClass<int> my_float_object(10);
12     MyClass<double> my_double_object(10.0);
13     return 0;
14 }
```

Template Parameters

```
1 template <typename T, size_t N = 10>
2 T AccumulateVector(const T& val) {
3     std::vector<T> vec(val, N);
4     return std::accumulate(vec.begin(), vec.end(), 0);
5 }
```

- Every **template** is parameterized by one or more **template parameters**:
template < parameter-list > declaration
- Think the **template parameters** the same way as any **function arguments**, but at **compile-time**.

Template Parameters

```
1 template <typename T, size_t N = 10>
2 T AccumulateVector(const T& val) {
3     std::vector<T> vec(val, N);
4     return std::accumulate(vec.begin(), vec.end(), 0);
5 }
6
7 using namespace std;
8 int main() {
9     cout << AccumulateVector(1) << endl;
10    cout << AccumulateVector<float>(2) << endl;
11    cout << AccumulateVector<float, 5>(2.0) << endl;
12    return 0;
13 }
```

Type Deduction

Type deduction for function templates:

```
1 #include <cstdio>
2
3 template <typename T>
4 void foo(T x) {
5     puts(__PRETTY_FUNCTION__);
6 }
7
8 int main() {
9     foo(4);           // void foo(T) [T = int]
10    foo(4.2);        // void foo(T) [T = double]
11    foo("hello");    // void foo(T) [T = const char *]
12 }
```

Type Deduction Rules (short)

- Each function parameter may contribute (or not) to the deduction of each template parameter (or not).
- At the end of this process, the compiler checks to make sure that each template parameter has been deduced at least once (otherwise: `couldn't infer template argument T`) and that all deductions agree with each other (otherwise: `deduced conflicting types for parameter T`).

Type Deduction

Type deduction for function templates:

```
1 template <typename T, typename U>
2 void f(T x, U y) {
3     // ..
4 }
5 template <typename T>
6 void g(T x, T y)
7 // ..
8 }
9
10 int main() {
11     f(1, 2); // void f(T, U) [T = int, U = int]
12     f(1, 2u); // void f(T, U) [T = int, U = unsigned int]
13     g(1, 2); // void g(T, T) [T = int]
14     g(1, 2u); // error: no matching function for call
15                     // to g(int, unsigned int)
16 }
```

Type Deduction

Type deduction for class templates:

```
1 template <typename T>
2 struct Foo {
3     public:
4         Foo(T x) : x_(x) {}
5         T x_;
6     };
7
8     int main() {
9         auto obj = Foo<int>(10).x_;
10        auto same_obj = Foo(10).x_;
11        auto vec = std::vector<int>{10, 50};
12        auto same_vec = std::vector{10, 50};
13 }
```

Note: New in C++17

Type Deduction Puzzle

```
1 template <typename T, typename U>
2 void foo(std::array<T, sizeof(U)> x,
3           std::array<U, sizeof(T)> y) {
4     puts(__PRETTY_FUNCTION__);
5 }
6
7 int main() {
8     foo(std::array<int, 8>{}, std::array<double, 4>{});
9     foo(std::array<int, 9>{}, std::array<double, 4>{});
10 }
```

Template Full Specialization

```
1 template <typename T>
2 bool is_void() {
3     return false;
4 }
5
6 template <>
7 bool is_void<void>() {
8     return true;
9 }
10
11 int main() {
12     std::cout << std::boolalpha
13             << is_void<int>() << std::endl
14             << is_void<void>() << std::endl;
15 }
```

Template Full Specialization

- Prefix the definition with `template<>`
- Then write the function **definition**.
- Usually means you don't need to write any more angle brackets at all.
- Unless `T` **can't** be deduced:

```
1 template <typename T>
2 int my_sizeof() {
3     return sizeof(T);
4 }
5
6 template <>
7 int my_sizeof<void>() {
8     return 1;
9 }
```

Template Full Specialization

- Prefix the definition with `template<>`
- Then write the function **definition**.
- Usually means you don't need to write any more angle brackets at all.
- Unless `T` can't be deduced/**defaulted**:

```
1 template <typename T = void>
2 int my_sizeof() {
3     return sizeof(T);
4 }
5
6 template <>
7 int my_sizeof() {
8     return 1;
9 }
```

Template Partial Specialization

```
1 template <typename T>
2 constexpr bool is_array = false;
3
4 template <typename Tp>
5 constexpr bool is_array<Tp[]> = true;
6
7 int main() {
8     std::cout << std::boolalpha;
9     std::cout << is_array<int>    << std::endl    // false
10        << is_array<int[]> << std::endl;    // true
11 }
```

A partial specialization is any specialization that is, itself, a template. It still requires further “customization” by the user before it can be used.

Template headers/source

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

<http://www.drdobbs.com/moving-templates-out-of-header-files/184403420>

Static code generation with `constexpr`

```
1 #include <iostream>
2 constexpr int factorial(int n) {
3     // Compute this at compile time
4     return n <= 1 ? 1 : (n * factorial(n - 1));
5 }
6
7 int main() {
8     // Guaranteed to be computed at compile time
9     return factorial(10);
10 }
```

- `constexpr` specifies that the value of a variable or function can appear in constant expressions

It only works if the variable of function **can** be defined at **compile-time**:

```
1 #include <array>
2 #include <vector>
3
4 int main() {
5     std::vector<int> vec;
6     constexpr size_t size = vec.size();    // error
7
8     std::array<int, 10> arr;
9     constexpr size_t size = arr.size();    // works!
10 }
```

error: constexpr variable 'size' must be initialized by a constant expression

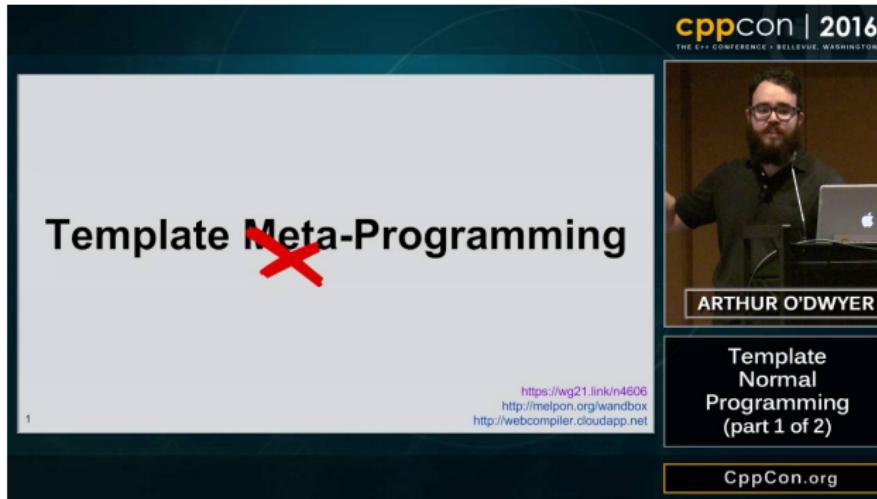
It only works if the variable of function **can** be defined at **compile-time**:

```
1 #include <array>
2 #include <vector>
3
4 int main() {
5     std::vector<int> vec;
6     constexpr size_t size = vec.size();    // error
7
8     std::array<int, 10> arr;
9     constexpr size_t size = arr.size();    // works!
10 }
```

error: constexpr variable 'size' must be initialized by a constant expression

Suggested Video

Template Normal Programming



<https://youtu.be/vwrXHznaYLA>

References

- <https://en.cppreference.com/w/cpp/language/templates>
- https://en.cppreference.com/w/cpp/language/function_template
- https://en.cppreference.com/w/cpp/language/class_template
- https://en.cppreference.com/w/cpp/language/template_parameters
- https://en.cppreference.com/w/cpp/language/template_argument_deduction
- https://en.cppreference.com/w/cpp/language/template_specialization
- https://en.cppreference.com/w/cpp/language/partial_specialization

Tools

- [GNU/Linux \[Tutorial\]](#)
 - Filesystem
 - Terminal
 - standard input/output
- [Text Editor](#)
 - Configuring
 - Terminal
 - Compile
 - Debug
- [Build systems](#)
 - `#include` statements
 - headers/sources
 - Libraries
 - Compilation flags
 - CMake
 - 3rd party libraries
- [Git \[Tutorial\]](#)
- [Homework submissions \[Tutorial\]](#)
- [Gdb \[Tutorial\]](#)
- [Web-based tools](#)
 - Quick Bench
 - Compiler Explorer
 - Cpp insights
 - Cppreference.com
- [Clang-tools \[Tutorial\]](#)
 - Clang-format
 - Clang-tidy
 - Clangd
 - Cppcheck
- [Google test \[tutorial\]](#)
- [OpenCV \[tutorial\]](#)

Core C++

- C++ basic syntax
- Variables
- Operators
- Scopes
- Built-in types
- Control structures (if, for, while)
- streams
- Input parameters
- C++ strings
- Functions
- Function overloading
- Pass by value / Pass by reference
- Namespaces
- Containers
- Iterators
- STL Algorithms
- Exceptions
- Utilities
- filesystem
- I/O Files
- Classes introduction

Modern C++

- enum classes
- Operator overloading
- Const correctness
- `typedef`/`using`
- static variables /methods
- Move Semantics
- Special Functions
- Inheritance
- Function Overriding
- Abstract classes
- Interfaces
- Strategy Pattern
- Singleton Pattern
- Polymorphism
- Typecasting
- Memory management
- Stack vs Heap
- Pointers
- `new/delete`
- `this` pointer
- Memory issues
- RAII
- Smart pointers
- Generic programming
- Template functions
- Template classes
- Template argument deduction
- Template partial specialization
- Template parameters
- `constexpr`
- Static code generation

Where to go from now on?

#	Date	Topics	Homework	Recommended Deadline	Official Deadline
Part I: C++ tools					
-	6-Apr	[[No Lectures]]	-	-	-
0	13-Apr	Course Introduction, Organization, Hello world	-	-	-
1	20-Apr	C++ Tools	Homework 1	1-May	8-May
Part II: The C++ core language					
2	27-Apr	C++ Basic syntax	Homework 2	8-May	15-May
3	4-May	C++ Functions	Homework 3	15-May	22-May
4	11-May	C++ STL	Homework 4	22-May	29-May
5	18-May	Filesystem + BoW Introduction	Homework 5	29-May	5-Jun
Part III: Modern C++					
6	25-May	Classes	Homework 6	5-Jun	12-Jun
7	1-Jun	OOP	Homework 7	12-Jun	19-Jun
8	8-Jun	Memory Management	Homework 8	19-Jun	26-Jun
9	15-Jun	Generics Programming	Homework 9	26-Jun	3-Jul
Part IV: Final Project "Place recognition using Bag of Visual Words in C++"					
10	22-Jun	Bag of Visual Words			
11	29-Jun				
12	6-Jul	[[No Lectures]]	Final Project		31 of July
13	13-Jul				