

Robotic Mapping & Localization

Kaveh Fathian

Assistant Professor

Computer Science Department

Colorado School of Mines

Lab04: C++

*Courtesy of Ignacio Vizzo, Igor Bogoslavskyi, Cyrill Stachniss

Lecture Outline

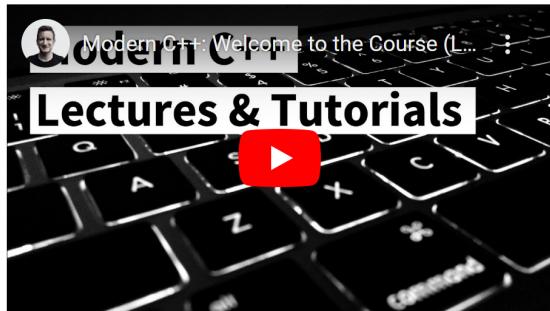
- **Introduction to C++**
 - A quick overview of many topics!



Suggested C++ Lectures

Modern C++ Course: Lectures & Tutorials (2020)

This is the Modern C++ course taught in 2020 at our lab plus useful tutorial videos that should support learning C++. Enjoy!



Modern C++: Welcome to the Course (Lectures and Tutorials, 2020 Edition)



Modern C++: The Basics (Lecture 0, I. Vizzo, 2020)

Modern C++ Course, Lecture 00: The Basics

Slides, Tutorials, and more:

—
Slides: [Link](#)

Related tutorials for this lecture:

- Bash Introduction: [Link](#)
- Command Line Pipes and redirection: [Link](#)

Information about this course: [Link](#)

<https://www.ipb.uni-bonn.de/index.html%3Fp=4044.html>

CPP-00 Modern C++: Course Introduction and Hello World (2018, Igor)



CPP-00 Modern C++: Course Introduction and Hello World (2018, Igor)



CPP-01 Modern C++: Variables, Basic Types, Control Structures (2018, Igor)



CPP-02 Modern C++: Compilation, Debugging, Functions, Header/Source, Libraries, CMake (2018, Igor)



CPP-03 Modern C++: Google Test, Namespaces, Classes (2018, Igor)



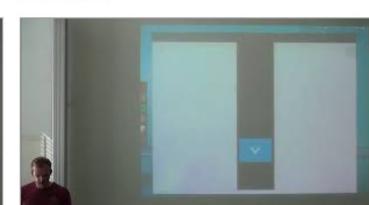
CPP-04 Modern C++: Move Semantics, Classes (2018, Igor)



CPP-05 Modern C++: Polymorphism, I/O, Stringstreams, CMake find (2018, Igor)



CPP-06 Modern C++: Static, Numbers, Arrays, Non-owning pointers, Classes (2018, Igor)



CPP-07 Modern C++: Pointers, const with pointers, Stack and Heap, Memory leaks (2018, Igor)



CPP-08 Modern C++: Smart/Unique/Shared ptrs, Associative con., Enumeration (2018, Igor)

<https://www.ipb.uni-bonn.de/modern-cpp/index.html>

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_f1; // var_f is valid within this scope
5 } // }<-this defines end of the scope
6
7 var_f1; // Error, var_f1 outside its scope
8
9 int var_f1; // Valid, var_f1 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

- C++ keywords have specific meanings and ***cannot*** be used as identifiers for variables, functions, or other user-defined names
- Attempting to do so will result in a compilation error.

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

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

```
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

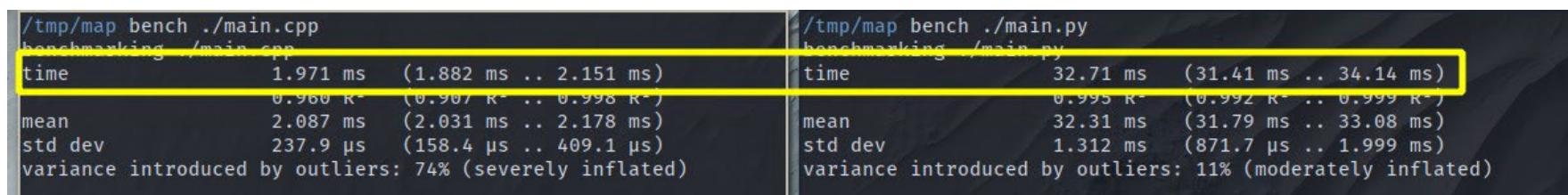
```
1 std::map<char, int> my_dict{{'a', 27}, {'b', 3}};
2 for (const auto& [key, value] : my_dict) {
3     cout << key << " has value " << value << endl;
4 }
```

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 \approx 15 times faster than Python



```
/tmp/map bench ./main.cpp
benchmarking ./main.cpp
time          1.971 ms  (1.882 ms .. 2.151 ms)
              0.960 R-  (0.907 R- .. 0.998 R-)
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 R-  (0.992 R- .. 0.999 R-)
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 carret_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 ???

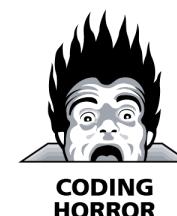
Some additional operations

- 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**

Google naming rules: https://google.github.io/styleguide/cppguide.html#General_Naming_Rules

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

- 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 }
```

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>"`

http://en.cppreference.com/w/cpp/language/main_function

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 parameteres 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 }
```

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.*

Definition taken from: <https://www.webopedia.com/TERM/F/function.html>

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.*

Extract from: Section 12 of "The C++ Programming Language Book by Bjarne Stroustrup"

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

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

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 defintion of f(), thus, this implementation  
4     int local_variable = 50;  
5 }  
6  
7 // at this point local_variable has not been intialized  
8 // is not accessiblly by any other part of the program  
9  
10 f(); //< When enter the function call, gets intialized
```

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

- 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.

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
- **GOOGLESTYLE** Only use them when readability gets much better
- **NACHOSTYLE** Never use them

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

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.
```



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. Is 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

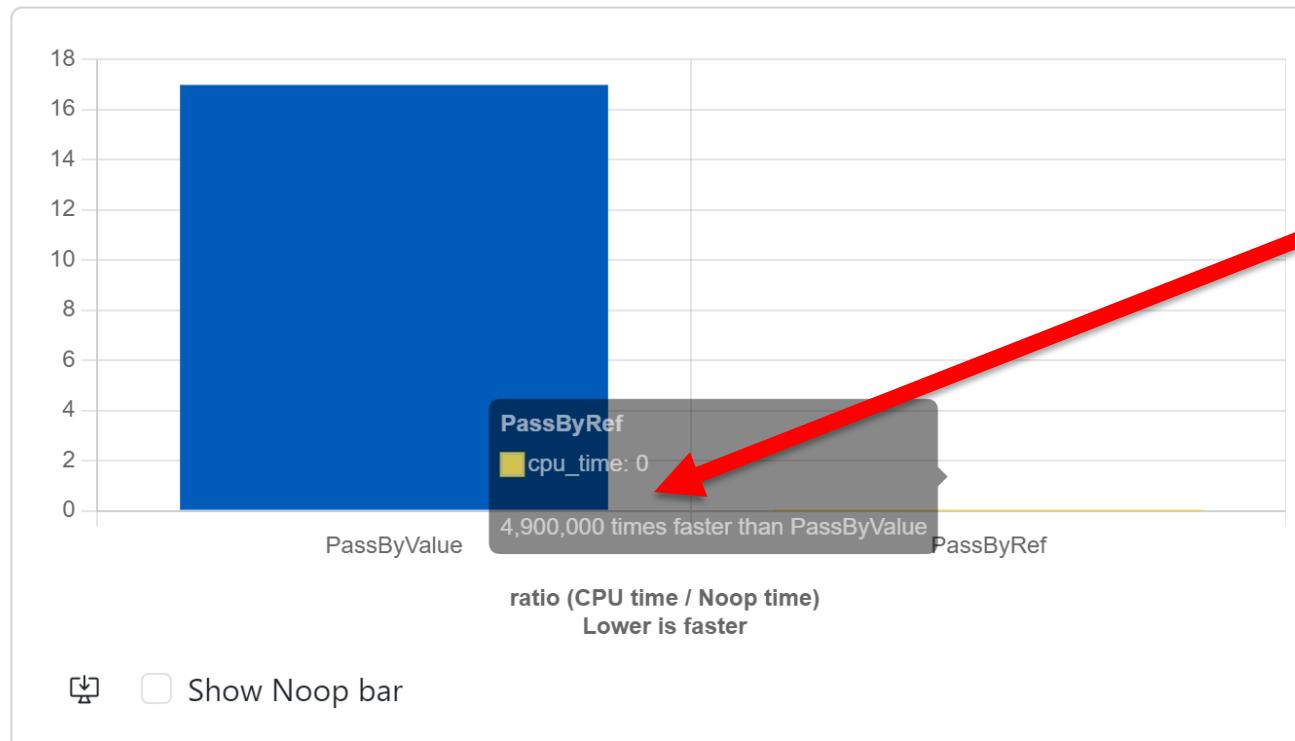
<http://quick-bench.com/LqwBICOM3KrQE4tqupBtzqJmCd>

Cost of passing by value

compiler = GCC 7.3 ▾ std = c++14 ▾ optim = O3 ▾
STL = libstdc++(GNU) ▾ Other flags ▾

 Run Benchmark Disassembly = None ▾ Clear cached results



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); }
```

Cehck 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 }
```

Cehck 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 }
```

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

Good Practices

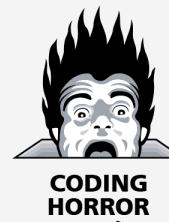
- Break up complicated computations into meaningful chunks and name them.
- Keep the length of functions **small** enough.
- Avoid unnecessary comments.
- One function should achieve **ONE** task.
- If you can't pick a short name, then **split** functionality.
- Avoid macros.
 - If you must use it, use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

Namespaces

module1

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

module2

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

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

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

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 }
```

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

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::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::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

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 }
```

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

- **Container** is a holder object that stores a collection of other objects (its elements).
- Containers are implemented as class templates, which allows great flexibility in the types of elements
- Containers that are very commonly used: vector, array, map, ...

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?

- 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/array>

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
 -
- **value** can be any type, you name it.
- These are called dictionaries `dict` in Python.

<http://en.cppreference.com/w/cpp/container/map>

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`

http://en.cppreference.com/w/cpp/container/unordered_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

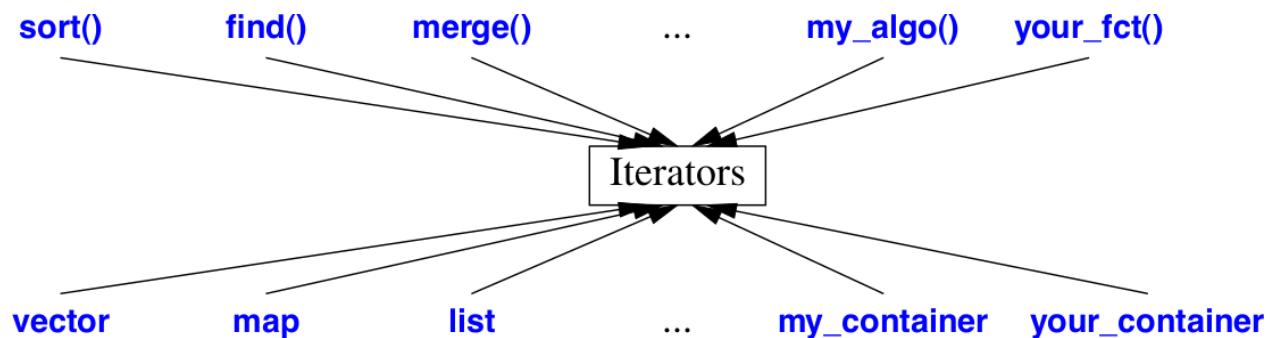
```
1 std::map<char, int> my_dict{{'a', 27}, {'b', 3}};  
2 for (const auto& [key, value] : my_dict) {  
3     cout << key << " has value " << value << endl;
```

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

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 `->` like 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).

Algorithms in standard library: <http://en.cppreference.com/w/cpp/algorithm>

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 proving 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 in t main() {
2     // value should be between [kMin,kMax]
3     const double kMax = 1.0 F;
4     const double kMin = 0.0 F;
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
```

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>
-

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

Reading and writing to files

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

```
1 #include <fstream>
2 using std::string;
3 using Mode = std::ios_base::openmode;
4
5 // ifstream: stream for input from file
6 std::ifstream f_in(string& file_name, Mode mode);
7
8 // ofstream: stream for output to file
9 std::ofstream f_out(string& file_name, Mode mode);
10
11 // stream for input and output to file
12 std::fstream f_in_out(string& file_name, Mode mode);
```

There are many modes under which a file can be opened

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

Regular columns

Use it when:

- The file contains organized data
- Every line has to have all columns

```
1 1 2.34 One 0.21
2 2 2.004 two 0.23
3 3 -2.34 string 0.22
```

O.K.

```
1 1 2.34 One 0.21
2 2 2.004 two 0.23
3 3 -2.34 string 0.22
```

Fail

```
1 1 2.34 One 0.21
2 2 2.004 two
3 3 -2.34 string 0.22
```

Reading from ifstream

```
1 #include <fstream> // For the file streams.  
2 #include <iostream>  
3 #include <string>  
4 using namespace std; // Saving space.  
5 int main() {  
6     int i;  
7     double a, b;  
8     string s;  
9     // Create an input file stream.  
10    ifstream in("test_cols.txt", ios_base::in);  
11    // Read data, until it is there.  
12    while (in >> i >> a >> s >> b) {  
13        cout << i << ", " << a << ", "  
14        << s << ", " << b << endl;  
15    }  
16    return (0);  
17 }
```

Reading files one line at a time

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

```
1 =====
2 HEADER
3 a = 4.5
4 filename = /home/ivizzo/.bashrc
5 =====
6 2.34
7 1 2.23
8 ER SIE ES
```

```
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
```

Class Basics

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

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

Class Basics

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

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

Class Basics

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

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

Class Basics

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

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

Example class definition

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

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

<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c-classes-and-class-hierarchies>

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

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

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

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

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
- **GOOGLESTYLE** All data must be **private**
- **GOOGLESTYLE** 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;
4                                         ^
```

Intuition lvalues, rvalues

- Every expression is an **lvalue** or an **rvalue**
- An **lvalue** is an expression that refers to an object in memory, typically identified by its address.
- Examples of **lvalues** are variables, array elements, and dereferenced pointers.
- An **rvalue** is an expression that does not have a specific memory location & is temporary
- Examples of **rvalues** include literals, temporary values, and the result of expressions

Intuition lvalues, rvalues

- 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.

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

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`.

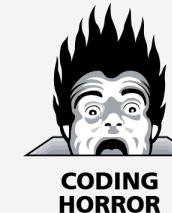
Hands on example

```
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;
```



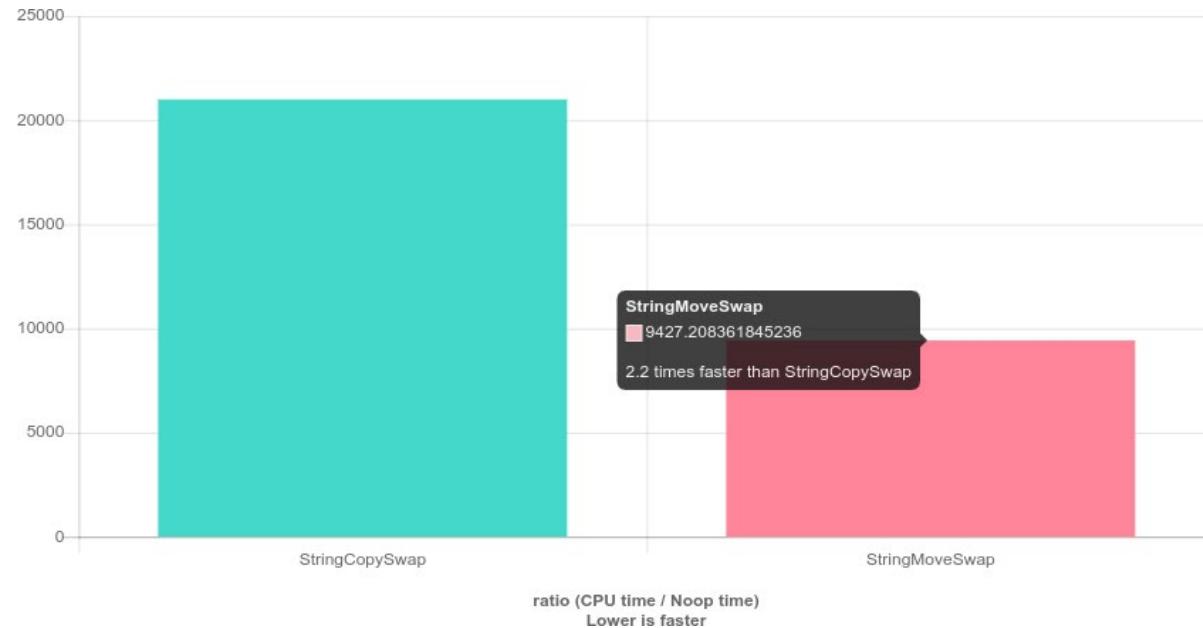
std::move performance

```
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", "names", "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 }
```

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     };
```

Arne Mertz: <https://arne-mertz.de/2015/02/the-rule-of-zero-revisited-the-rule-of-all-or-nothing/>
<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#cdefop-default-operations>

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 too 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 }
```

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`, ...
- **GOOGLESTYLE** Name enum type as other types, `CamelCase`
- **GOOGLESTYLE** 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}
17int 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,   // Hexacedimal.  
4     OPTION_3 = 13  
5 };
```

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

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

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     };
```

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** Use it only when improves only when improves code performance/readability.

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

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)

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!

Full specs: http://en.cppreference.com/w/cpp/language/static_cast

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

Full specs: http://en.cppreference.com/w/cpp/language/dynamic_cast

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

Full specs: http://en.cppreference.com/w/cpp/language/reinterpret_cast

const_cast

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

Full specs: http://en.cppreference.com/w/cpp/language/const_cast

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.

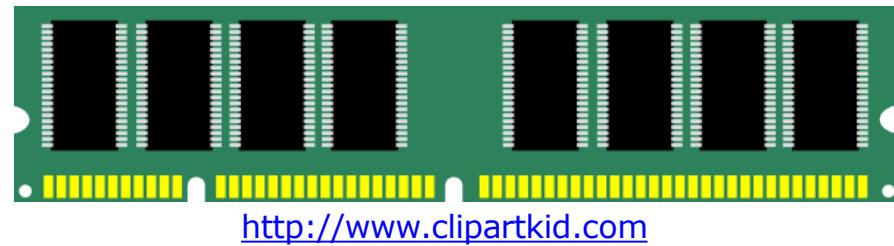
Google Style

- **GOOGLESTYLE** 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.

Working memory or RAM



- Working memory has **linear addressing**
- Every byte has an **address** usually presented in hexadecimal form,
e.g. **0x7ffb7335fdc**
- 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

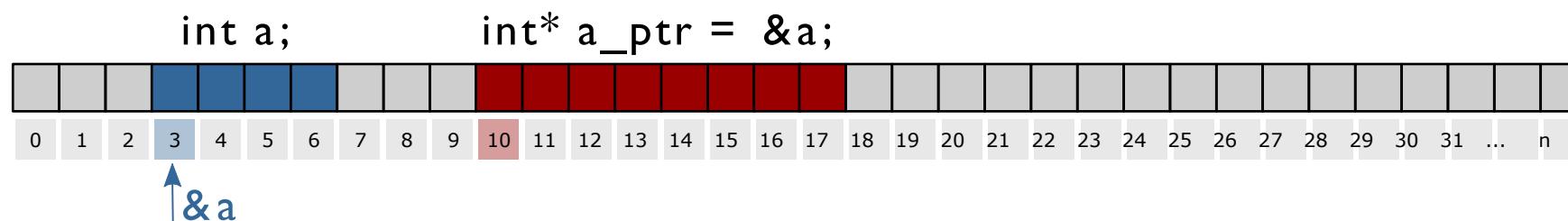
- Memory pointed to by a raw pointer is not removed when pointer goes out of scope
- Pointers can either own memory or not
- Owning memory means being responsible for its cleanup
- Raw pointers should never own memory
- We will talk about smart pointers that own memory later

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;
```

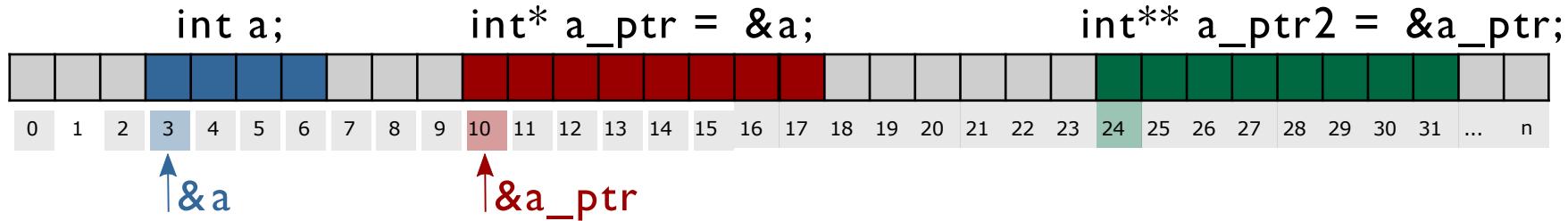


<http://www.cplusplus.com/doc/tutorial/pointers/>

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 = , b =
2 a = , b =
```

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 }
```

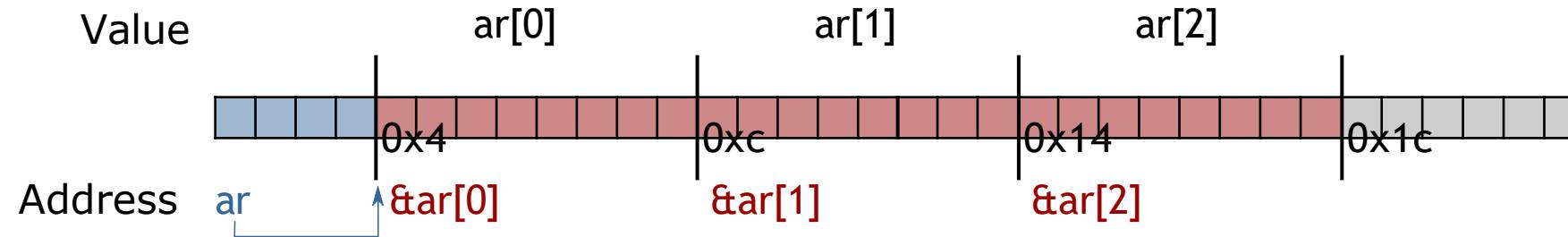
```
1 ptr address: 0x400830
2 value under ptr: -1991643855
3 new ptr address: 0
4 ptr size: 8 bytes (64bit)
```

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 }
```

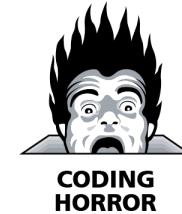
Arrays in memory and pointers



- 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

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



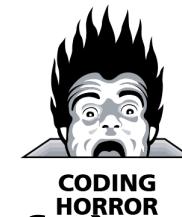
```
1 #include <stdio.h>
2 int main(int argc, char const* argv[]) {
3     int size = 2;
4     int* ptr = nullptr;
5     {
6         int ar[size];
7         ar[0] = 42;
8         ar[1] = 13;
9         ptr = ar;
10    }
11    for (int i = 0; i < size; ++i) {
12        printf("%d\n", ptr[i]);
13    }
14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

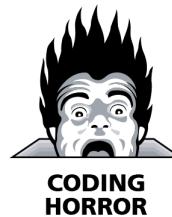


Operators `new` and `new[]`

- 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

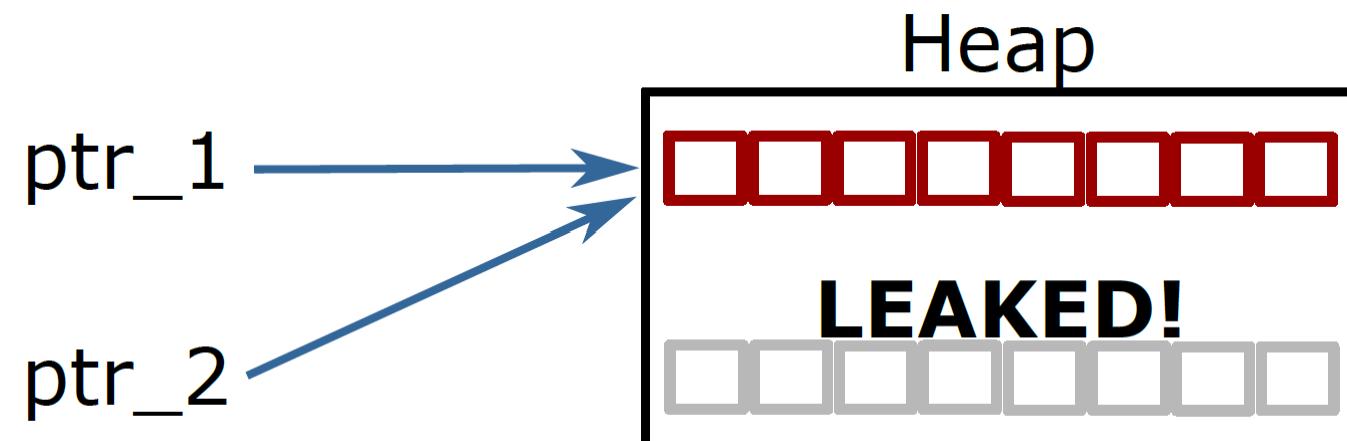
```
1 #include <iostream>
2 using std::cout; using std::endl;
3 int main() {
4     int size = 2; int* ptr = nullptr;
5     {
6         ptr = new int[size];
7         ptr[0] = 42; ptr[1] = 13;
8     } // End of scope does not free heap memory!
9 // Correct access, variables still in memory.
10 for (int 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

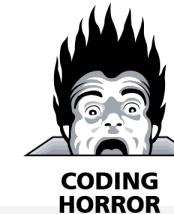
- Can happen when working with Heap memory if we are not careful
- Memory leak: memory allocated on Heap access to which has been lost

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 (delete)

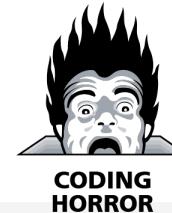


```
1 int main() {
2     int *ptr_1 = nullptr;
3     int *ptr_2 = nullptr;
4
5     // Allocate memory for two bytes on the heap.
6     ptr_1 = new int;
7     ptr_2 = new int;
8     cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
9
10    // Overwrite ptr_2 and make it point where ptr_1
11    ptr_2 = 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
   x0000000010a3010 ***
```

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



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

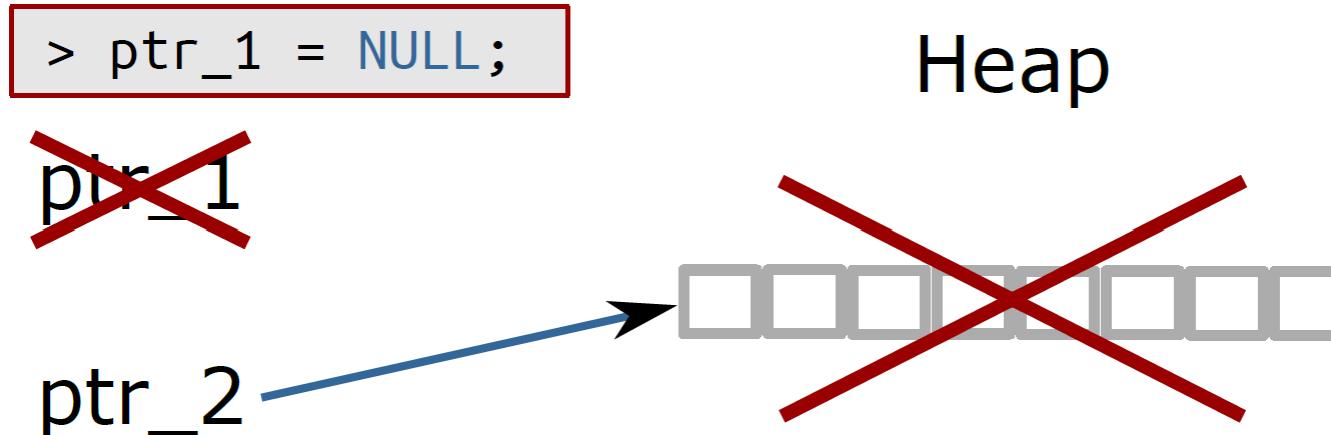
```
1 # ...
2 Iteration: 19 done. 20 GiB has been allocated!
3 Iteration: 20 done. 21 GiB has been allocated!
4 Iteration: 21 done. 22 GiB has been allocated!
5 Iteration: 22 done. 23 GiB has been allocated!
6 terminate called after throwing an instance of 'std::
   bad_alloc'
7   what(): std::bad_alloc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

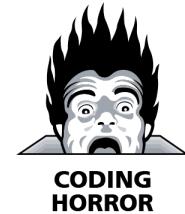
Dangling pointer

```
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

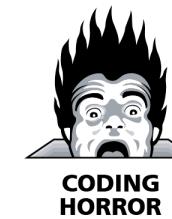
- Dangling Pointer: pointer to a freed memory
- Think of it as the opposite of a memory leak
- Dereferencing a dangling pointer causes undefined behavior



Dangling pointer example

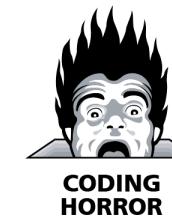
```
1 #include <iostream>
2 using std::cout; using std::endl;
3 int main() {
4     int size = 5;
5     int *ptr_1 = new int[size];
6     int *ptr_2 = ptr_1; // Point to same data!
7     ptr_1[0] = 100; // Set some data.
8     cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
9     cout << "ptr_2[0]: " << 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: 0x0000000000877c20 ***
```

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!

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)?

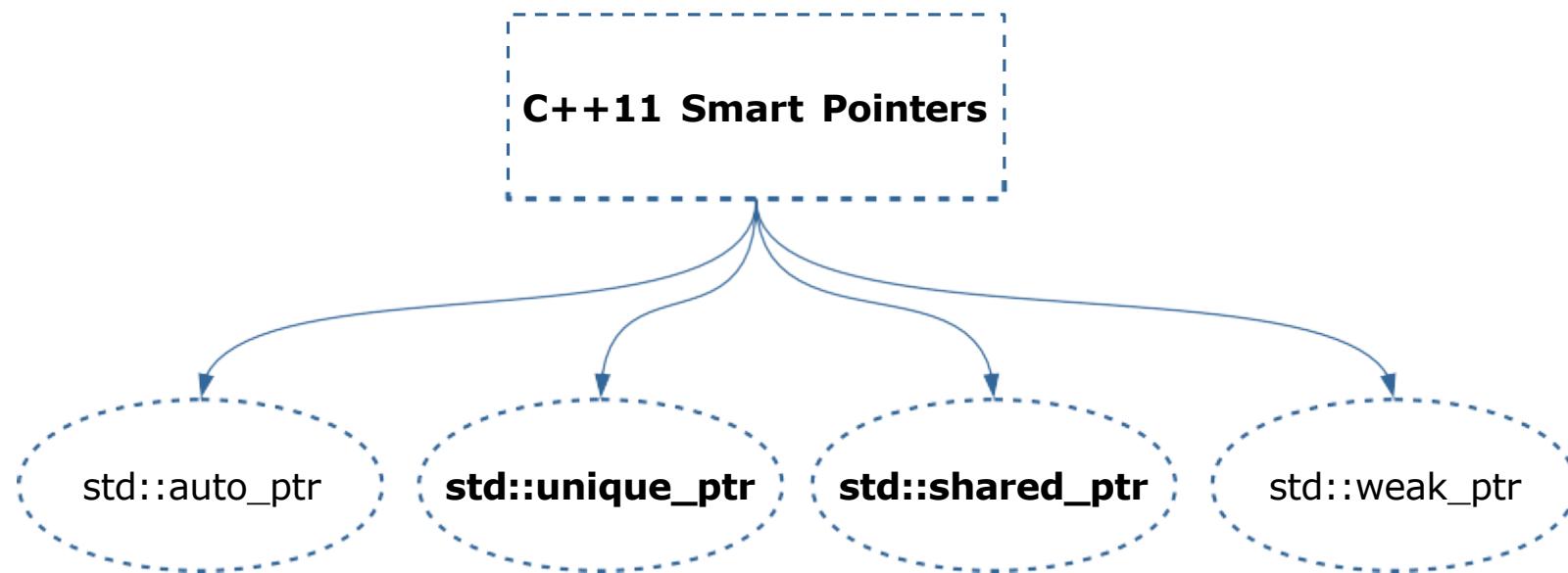
Raw pointers are hard to love

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!

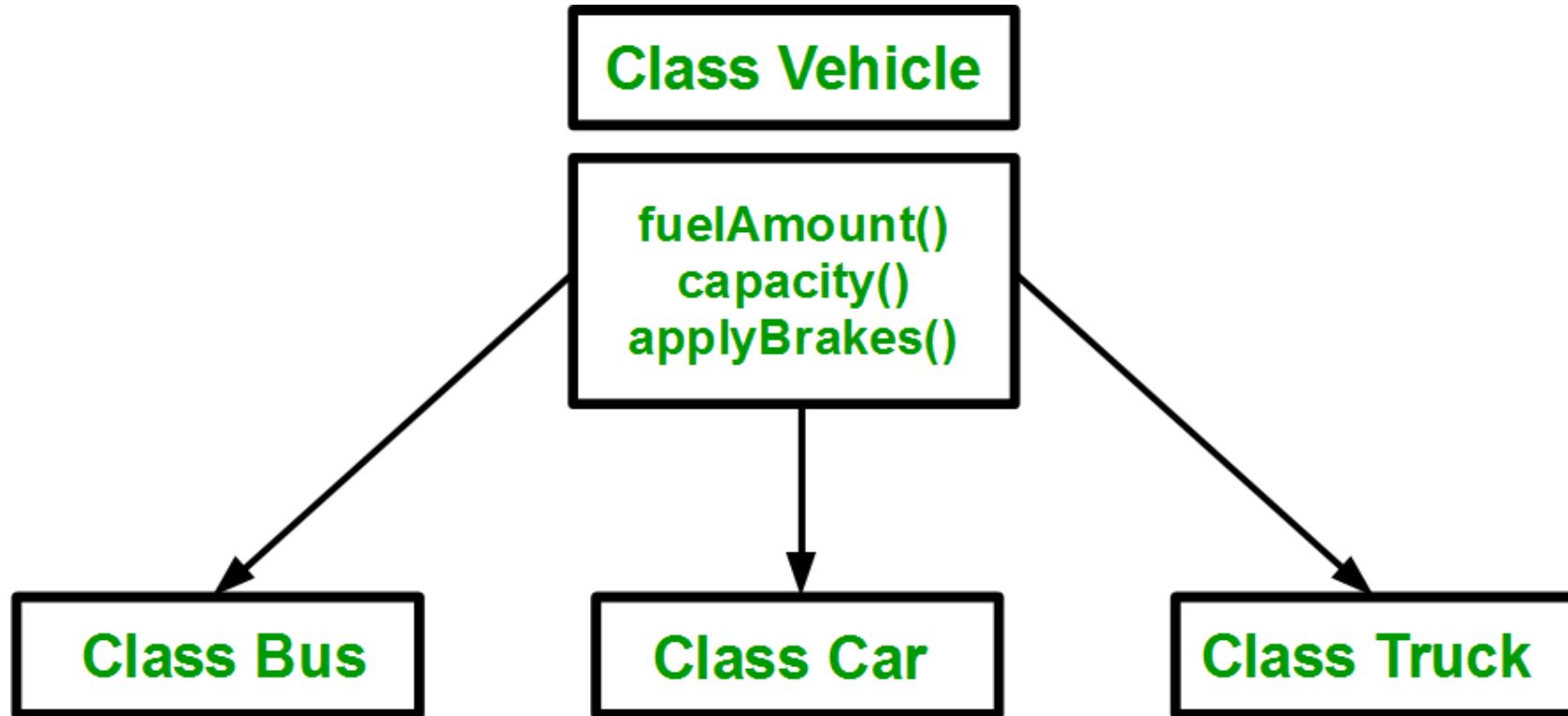
Smart pointers apart from memory allocation behave exactly as raw pointers:

- 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 us 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
```

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>);
```

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

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_ptrs` 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 }
```

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 }
```

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.h** file, implement in **NAME_impl.h** file, add `#include <NAME_impl.h>` in the end of **NAME.h**
 3. Declare in ***.h** 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

References

- Modern C++ for Computer Vision:
 - <https://www.ipb.uni-bonn.de/cpp-course-2020/index.html>
 - <https://www.ipb.uni-bonn.de/teaching/modern-cpp/index.html>
- Google Code Styleguide: <https://google.github.io/styleguide/cppguide.html>
- GCC Manual: <https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/>