Modern C++ for Computer Vision and Image Processing

Lecture 6: Modern C++ Classes

Ignacio Vizzo and Cyrill Stachniss

UNIVERSITÄT BONN

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 **O**bject **O**riented **P**rogramming language
- `string`, `vector`, etc. are all classes

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

C++ Class Anatomy

class.cpp class.cpp

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

Class Definition

Constructors and Destructors

Member Functions

Operators

Data Members

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

```cpp
#include <iostream>
#include <string>
struct NamedInt {
  int num;
  std::string name;
};

void PrintStruct(const NamedInt& s) {
  std::cout << s.name << " " << s.num << std::endl;
}

int main() {
  NamedInt var{1, std::string{"hello"}};
  PrintStruct(var);
  PrintStruct({10, std::string{"world"}});
  return 0;
}
```

Data stored in a class

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

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

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

Constructors and Destructor

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

Many ways to create instances

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

Setting and getting data

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

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

Declaration and definition

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

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

Always initialize members for classes

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

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

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

Classes as modules

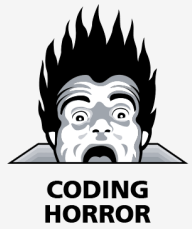
- Prefer encapsulating information that belongs together into a class
- **Separate declaration and definition** of the class into header and source files
- Typically, class `SomeClass` is declared in `some_class.hpp` and is defined in `some_class.cpp`

Const correctness

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

Typical const error

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




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

Intuition lvalues, rvalues

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

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

std::move

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

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

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;
```

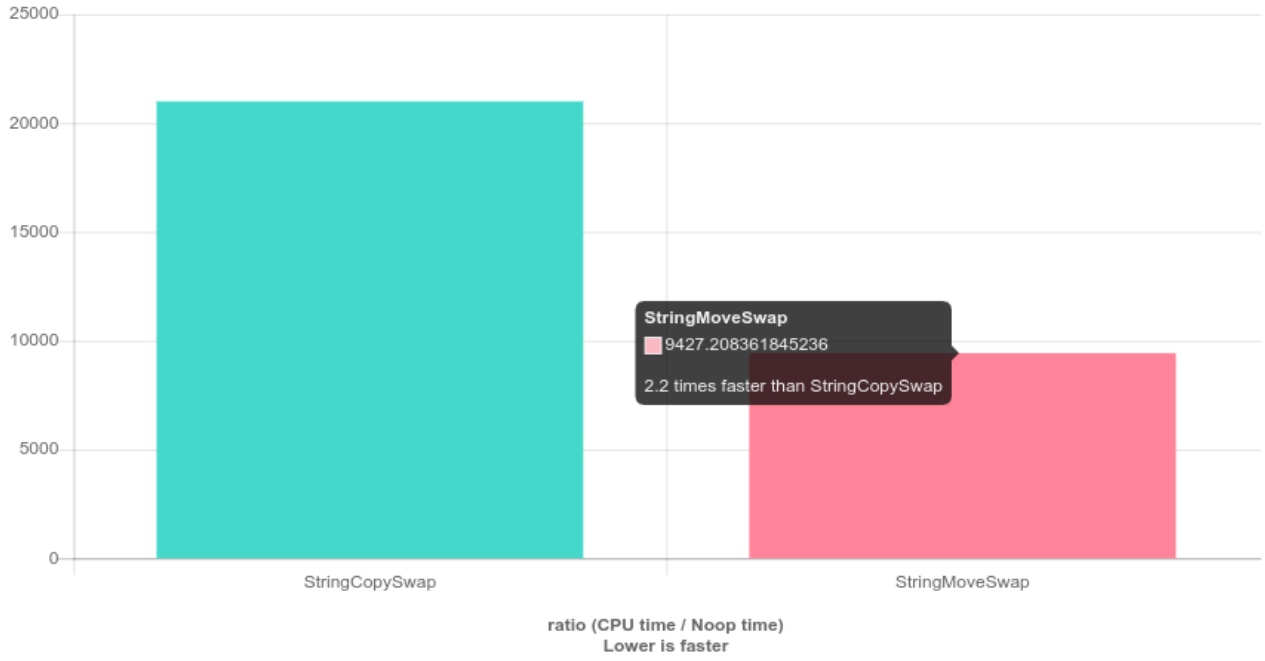
`std::move` performance

```cpp
// MyClass has a private member that contains 200 strings
struct MyClass {
  int id_ = 0;
  std::vector<std::string> names_{
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "names", "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name",
      "name", "name",  "name", "name", "name", "name", "name", "name", "name"};
};
```

`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 <

```cpp
#include <algorithm>
#include <vector>
class Human {
 public:
  Human(int kindness) : kindness_{kindness} {}
  bool operator<(const Human& other) const {
    return kindness_ < other.kindness_;
  }

 private:
  int kindness_ = 100;
};
int main() {
  std::vector<Human> humans = {Human{0}, Human{10}};
  std::sort(humans.begin(), humans.end());
  return 0;
}
```

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 **L**value
- 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.
```

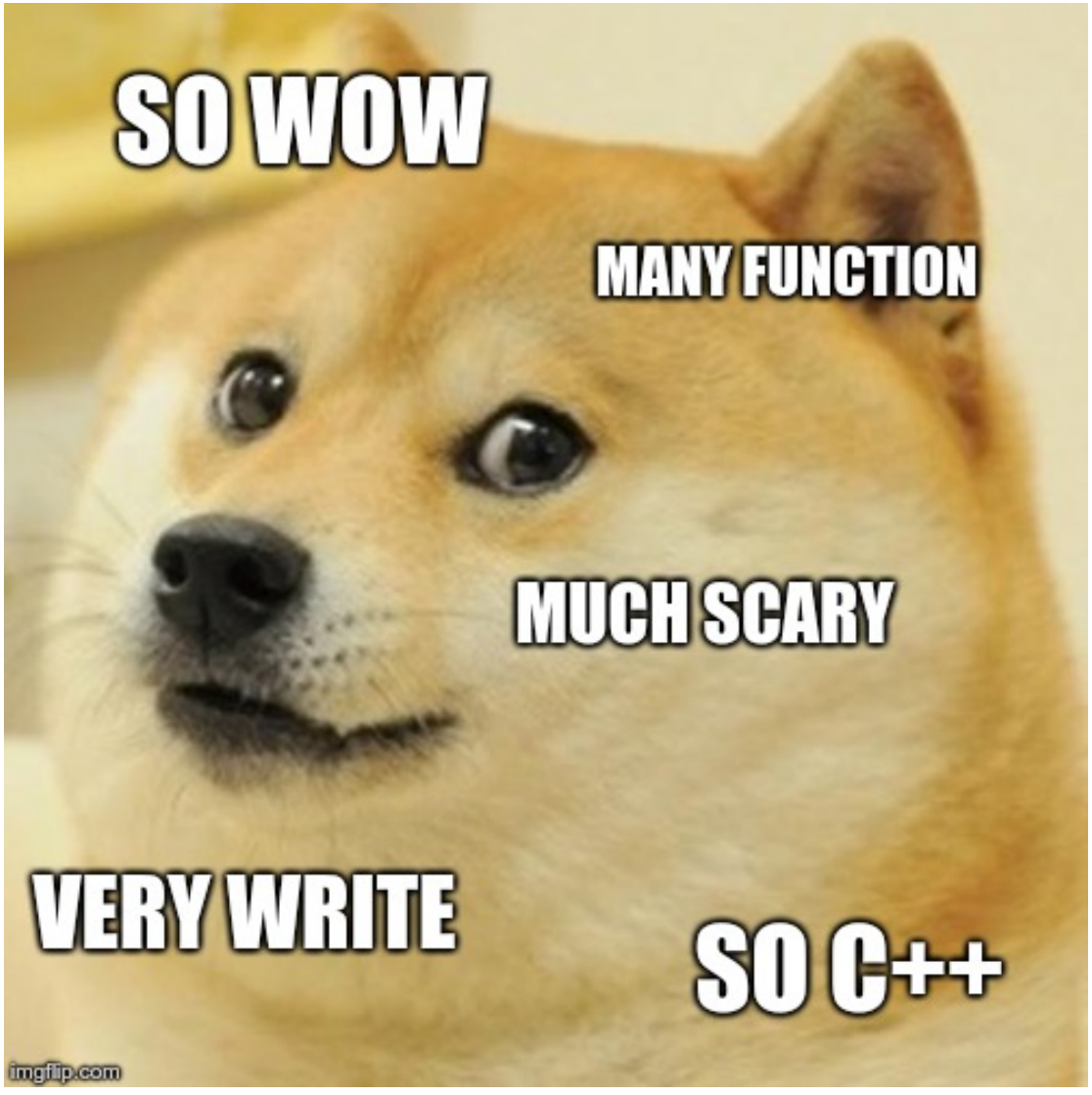
Move assignment operator

- **Called automatically** when the object is **assigned a new value** from an **R**value
- 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.
```

```cpp
class MyClass {
 public:
  MyClass() { cout << "default" << endl; }
  // Copy(&) and Move(&&) constructors
  MyClass(const MyClass& other) {
    cout << "copy" << endl;
  }
  MyClass(MyClass&& other) {
    cout << "move" << endl;
  }
  // Copy(&) and Move(&&) operators
  MyClass& operator=(const MyClass& other) {
    cout << "copy operator" << endl;
  }
  MyClass& operator=(MyClass&& other) {
    cout << "move operator" << endl;
  }
};

int main() {
  MyClass a;                  // Calls DEFAULT constructor
  MyClass b = a;              // Calls COPY constructor
  a = b;                      // Calls COPY assignment operator
  MyClass c = std::move(a); // Calls MOVE constructor
  c = std::move(b);           // Calls MOVE assignment operator
}
```


SO WOW
MANY FUNCTION
MUCH SCARY
VERY WRITE
SO C++
imgflip.com

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

```
#include <iostream>
using std::cout;
using std::endl;

// Include the Counted class declaration and
// Initialize the static member of the class only once.
// This could be any value
#include "Counted.hpp"
int Counted::count = 0;

int main() {
  Counted a, b;
  cout << "Count: " << Counted::count << endl;
  Counted c;
  cout << "Count: " << Counted::count << endl;
  return 0;
}
```

```cpp
#include <cmath>

class Point {
 public:
  Point(int x, int y) : x_(x), y_(y) {}

  static float Dist(const Point& a, const Point& b) {
    int diff_x = a.x_ - b.x_;
    int diff_y = a.y_ - b.y_;
    return sqrt(diff_x * diff_x + diff_y * diff_y);
  }

  float Dist(const Point& other) {
    int diff_x = x_ - other.x_;
    int diff_y = y_ - other.y_;
    return sqrt(diff_x * diff_x + diff_y * diff_y);
  }

 private:
  int x_ = 0;
  int y_ = 0;
};
```

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 }
```

Using for type aliasing

- Use word `using` to declare new types from existing and to create type aliases
- **Basic syntax:** `using NewType = OldType;`
- `using` is a versatile word
- When used outside of functions declares a new type alias
- When used in function creates an alias of a type available in the current scope
- http://en.cppreference.com/w/cpp/language/type_alias

Using for type aliasing

```
#include <array>
#include <memory>
template <class T, int SIZE>
struct Image {
  // Can be used in classes.
  using Ptr = std::unique_ptr<Image<T, SIZE>>;
  std::array<T, SIZE> data;
};
// Can be combined with "template".
template <int SIZE>
using Imagef = Image<float, SIZE>;
int main() {
  // Can be used in a function for type aliasing.
  using Image3f = Imagef<3>;
  auto image_ptr = Image3f::Ptr(new Image3f);
  return 0;
}
```

Enumeration classes

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

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

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

```cpp
#include <iostream>
#include <string>
using namespace std;
enum class Channel { STDOUT, STDERR };
void Print(Channel print_style, const string& msg) {
  switch (print_style) {
    case Channel::STDOUT:
      cout << msg << endl;
      break;
    case Channel::STDERR:
      cerr << msg << endl;
      break;
    default:
      cerr << "Skipping\n";
  }
}
int main() {
  Print(Channel::STDOUT, "hello");
  Print(Channel::STDERR, "world");
  return 0;
}
```

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 };
```

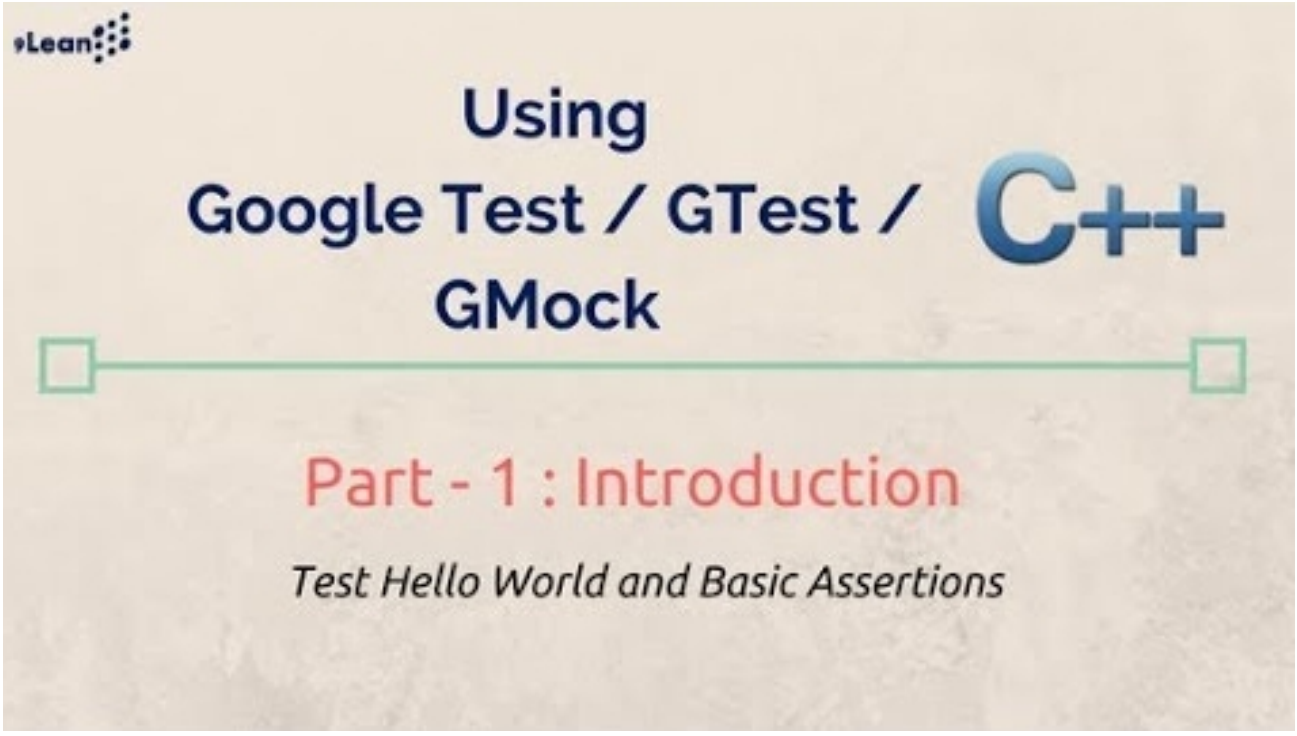
Suggested Video

C++ Classes

https://youtu.be/2BP8NhxjrO0

Suggested Video

Unit Tests



https://youtu.be/nbFXI9SDfbk

References 1

- **Classes**

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

- **Data Members**

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

- **Member Functions**

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

- **Static**

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

- **Operators**

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

References 2

- **Constructors**

  https://en.cppreference.com/w/cpp/language/constructor
- **Destructor**

  https://en.cppreference.com/w/cpp/language/destructor
- **Copy Constructor**

  https://en.cpprefeDestructorrence.com/w/cpp/language/copy_destructor
- **Move Constructor**

  https://en.cppreference.com/w/cpp/language/move_constructor
- **Copy Assignment**

  https://en.cppreference.com/w/cpp/language/copy_assignment
- **Move Assignment**

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