Modern C++ for Computer Vision and Image Processing

Lecture 6: Modern C++ Classes

Ignacio Vizzo and Cyrill Stachniss
Create new types with classes and structs

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

http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c-classes-and-class-hierarchies
C++ Class Anatomy

```cpp
class MyNewType {
public:
    MyNewType();
    ~MyNewType();

public:
    void MemberFunction1();
    void MemberFunction2() const;
    static void StaticFunction();

public:
    MyNewType &operator+=(const MyNewType &other);
    std::ostream &operator<<(std::ostream &os, const MyNewType &obj);

private:
    int a_;
    std::vector<float> data_;
    MyType2 member_;
};
```
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`:

```c
struct ExampleStruct {
  Type value;
  Type value;
  Type value;
  // No functions!
};
```

- `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

class SomeClass {
    public:
        SomeClass(); // Default constructor.
        SomeClass(int a); // Custom constructor.
        SomeClass(int a, float b); // Custom constructor.
        ~SomeClass(); // Destructor.
};

// How to use them?
int main() {
    SomeClass var_1; // Default constructor
    SomeClass var_2(10); // Custom constructor
    // Type is checked when using {} braces. Use them!
    SomeClass var_3{10}; // Custom constructor
    SomeClass var_4 = {10}; // Same as var_3
    SomeClass var_5{10, 10.0}; // Custom constructor
    SomeClass var_6 = {10, 10.0}; // Same as var_5
    return 0;
}
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

```cpp
class Student {
public:
  Student(int id, string name): id_{id}, name_{name} {}
  int id() const { return id_; }
  const string& name() const { return name_; }
private:
  int id_{};
  string name_{};
};
```
Declaration and definition

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

```cpp
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

```cpp
class Student {
public:
    // No need for default constructor.
    // Getters and functions omitted.
private:
    int earned_points_ = 0;
    float happiness_ = 1.0f;
};
```

**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

```cpp
#include <iostream>
#include <string>
using namespace std;

class Student {
  public:
    Student(string name) : name_(name) {}
    // This function *might* change the object
    const string& name() { return name_; }
  private:
    string name_; 
};

void Print(const Student& student) {
  cout << "Student: " << student.name() << endl;
}
```

error: passing "const Student" as "this" argument discards qualifiers [-fpermissive]
cout << "Student: " << student.name() << endl;
```
**Intuition lvalues, rvalues**

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

```cpp
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

```cpp
#include <iostream>
#include <string>
using namespace std; // Save space on slides.

void Print(const string& str) {
    cout << "lvalue: " << str << endl;
}

void Print(string&& str) {
    cout << "rvalue: " << str << endl;
}

int main() {
    string hello = "hi";
    Print(hello);
    Print("world");
    Print(std::move(hello));
    // DO NOT access "hello" after move!
    return 0;
}
```
Never access values after move

The value after move is undefined

```cpp
string str = "Hello";
vector<string> v;

// uses the push_back(const T&) overload, which means
// we'll incur the cost of copying str
v.push_back(str);
cout << "After copy, str is " << str << endl;

// uses the rvalue reference push_back(T&&) overload,
// which means no strings will be copied; instead,
// the contents of str will be moved into the vector.
// This is less expensive, but also means str might
// now be empty.
v.push_back(move(str));
cout << "After move, str is " << str << endl;
```
std::move performance

// 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", "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

```cpp
class MyClass {
    // MyClass implementation
};

// Copy swap
void copy_swap(MyClass& obj1, MyClass& obj2) {
    MyClass tmp = obj1; // copy obj1 to tmp
    obj1 = obj2; // copy obj2 to obj1
    obj2 = tmp; // copy tmp to obj1
}

// Move swap
void move_swap(MyClass& obj1, MyClass& obj2) {
    MyClass tmp = std::move(obj1); // move obj1 to tmp
    obj1 = std::move(obj2); // move obj2 to obj1
    obj2 = std::move(tmp); // move tmp to obj1
}
```
std::move performance

Quick Benchmark available to play:
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:
  
  [Link](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 `<<`

```cpp
#include <iostream>
#include <vector>

class Human {
    public:
        int kindness(void) const { return kindness_; } 
    private:
        int kindness_ = 100;
};

std::ostream& operator<<(std::ostream& os, const Human& human) {
    os << "This human is this kind: " << human.kindness();
    return os;
}

int main() {
    std::vector<Human> humans = {Human{0}, Human{10}};
    for (auto&& human : humans) {
        std::cout << human << std::endl;
    }
    return 0;
}
```
Copy constructor

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

```cpp
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

```cpp
MyClass a; // Calling default constructor.
MyClass b(a); // Calling copy constructor.
MyClass c = a; // Calling copy constructor.
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)`

```cpp
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 \texttt{R} value
- For class \texttt{MyClass} has a signature:
  \texttt{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.
```
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++
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

```cpp
class MyClass {
public:
    MyClass() = default;
    MyClass(MyClass&& var) = default;
    MyClass(const MyClass& var) = default;
    MyClass& operator=(MyClass&& var) = default;
    MyClass& operator=(const MyClass& var) = default;
};
```

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`
  ```cpp
  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:

  ```cpp
  ClassName::MethodName(<params>)
  ```
class Counted {
    public:
        // Increment the count every time someone creates
        // a new object of class Counted
        Counted() { Counted::count++; }

        // Decrement the count every time someone deletes
        // any object of class Counted
        ~Counted() { Counted::count--; }

        // Static counter member. Keep the count of how
        // many objects we've created so far
        static int count;
};

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

```cpp
#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

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

int main() {
    Point p1(2, 2);
    Point p2(1, 1);
    // Call the static method of the class Point
    cout << "Dist is " << Point::Dist(p1, p2) << endl;

    // Call the class-method of the Point object p1
    cout << "Dist is " << p1.Dist(p2) << endl;
}
```
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

```cpp
#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>
template class <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`

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

- Use values as:
  ```cpp
  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

```java
enum class EnumType {
    OPTION_1 = 10, // Decimal.
    OPTION_2 = 0x2, // Hexadecimal.
    OPTION_3 = 13
};
```
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**

- **Member Functions**
  https://en.cppreference.com/w/cpp/language/member_functions

- **Static**

- **Operators**
References 2

- **Constructors**
  

- **Destructor**
  

- **Copy Constructor**
  

- **Move Constructor**
  

- **Copy Assignment**
  

- **Move Assignment**
  