Modern C++ for Computer Vision and Image Processing

Igor Bogoslavskyi

Outline

Move semantics

Classes

- Operator overloading
- Making your class copyable
- Making your class movable
- Rule of all or nothing
- Inheritance

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
```

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  #include <iostream>
2  #include <string>
3  #include <vector>
4  using namespace std;  // Save space on slides.
5  int main() {
6    string hello = "hello";
7    vector<string> owner;
8    owner.emplace_back(hello);          // Copy.
9    owner.emplace_back(move(hello));    // Move.
10   cout << hello << endl;              // Undefined.
11   return 0;
12 }
```

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 <

```
#include <algorithm>
#include <vector>
using namespace std;
class Human {
 public:
  Human(int kindness) : kindness_{kindness} {}
  bool operator<(const Human& other) const {
    return kindness_ < other.kindness_;
  }
 private:
  int kindness_ = 100;
};
int main() {
  vector<Human> humans = {Human{0}, Human{10}};
  std::sort(humans.begin(), humans.end());
  return 0;
}
```

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
#include <iostream>
using std::cout; using std::endl;
class Hmm {
 public:
  Hmm() { cout << "default" << endl; }
  Hmm(const Hmm& other) { cout << "copy" << endl; }
  Hmm(Hmm&& other) { cout << "move" << endl; }
  Hmm& operator=(const Hmm& other) {
    cout << "copy operator" << endl; return *this;
  }
  Hmm& operator=(Hmm&& other) {
    cout << "move operator" << endl; return *this;
  }
};
int main() {
  Hmm a;
  Hmm b = a;
  a = b;
  Hmm c = std::move(a);
  c = std::move(b);
  return 0;
}
```

Do I need to define all of them?

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

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
- 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

Inheritance

- Classes and structs 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

```cpp
#include <iostream>
using std::cout; using std::endl;
class Rectangle {
 public:
  Rectangle(int w, int h) : width_{w}, height_{h} {}
  int width() const { return width_; }
  int height() const { return height_; }
 protected:
  int width_ = 0;
  int height_ = 0;
};
class Square : public Rectangle {
 public:
  explicit Square(int size) : Rectangle{size, size} {}
};
int main() {
  Square sq(10);  // Short name to save space.
  cout << sq.width() << " " << sq.height() << endl;
  return 0;
}
```

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

References

- **Fluent C++: structs vs classes:**
  https://goo.gl/NFo8HP [shortened]