C/C++: Lecture 3

Vorobev D.V

18.09.2020

OOP

Principles

- Encapsulation - it is a property of hiding data and part of functionality of *entity*$_1$ from *entity*$_2$ that operates with *entity*$_1$
- Inheritanhce - it is a property of setting hierarchy relation between *entities*
- Polymorphism - it is a property of an overriding general interface by special entities

class

- User type
- Describes designing *entity*
- Default access specifier *private*

struct

- User type
- Describes designing *entity*
- Defaullt access specifier *public*

class, struct

```cpp
#include <iostream>

class A {
    int x = 10;
};

int main() {
    A a;
    // CE
    std::cout << a.x;
    return 0;
}
```

```cpp
#include <iostream>

struct A {
    int x = 10;
};

int main() {
    A a;
    // 10
    std::cout << a.x;
    return 0;
}
```

Access specifiers

- public - we can access field or method outside the class
- private - we cannot access field or method outside the class
- protected - demonstrates its behavior in the context of inheritance (we will dig deeper some later)

class, struct

Encapsulation

An introduction of access specifiers allows to implement Encapsulation

```cpp
#include <iostream>

class A {
    private:
        int x = 10;
};

int main() {
    A a;
    // x сокрыт от внешнего доступа
    std::cout << a.x;
    return 0;
}
```

Constructor

Default constructor

- Parameter list: empty
- It is implicitly generated if there are no other constructor.

```
#include <iostream>

class A {};

int main() {
    //CE: нет
    A a;
    return 0;
}
```

Constructor

Copy constructor

- Parameter list: T& for the class type T
- It is called when *entity*$_1$ initializes *entity*$_2$
  Cases:
  - on initialization
  - on passing an object to function
  - on returning an object from function

```cpp
#include <iostream>

class A {
    public:
        A(const A& value) {}
};
```

Copy constructor: Intialization

```cpp
#include <iostream>

class A {
public:
    A(const A& value) {
        std::cout << "copy";
    }
};

int main() {
    A x;
    // "copy"
    A y = x;
    return 0;
}
```

Copy constructor: Passing

Constructor is called

```cpp
#include <iostream>

class A {
public:
    A() = default;
    A(const A& value) {
        std::cout << "copy";
    }
};
void foo(const A val) {};

int main() {
    A x;
    foo(x);
    return 0;
}
```

Constructor is not called

```cpp
#include <iostream>

class A {
public:
    A() = default;
    A(const A& value) {
        std::cout << "copy";
    }
};
void foo(const A& val) {};

int main() {
    A x;
    foo(x);
    return 0;
}
```

Overloading Constructor

```
#include <iostream>

struct A {
    A() {std::cout << 0;}
    A(int x) {std::cout << 1;}
    A(int x, int y) {std::cout << 2;}
    A(const A& x) {std::cout << 3;}
};

int main() {
    A a;
    A b(1);
    A c(1, 2);
    A d = a;
    // 0123
    return 0;
}
```

Name hiding

```cpp
#include <iostream>

struct A {
    int x = 10;
    void foo() {
        int x = 30;
        // 30
        std::cout << x;
    }
};

int main() {
    A a;
    a.foo();
    return 0;
}
```

```cpp
#include <iostream>

struct A {
    int x = 10;
    void foo() {
        // 10
        std::cout << x;
    }
};

int main() {
    A a;
    a.foo();

    return 0;
}
```

this

this

It is a pointer to an instance of the class type

```cpp
#include <iostream>

struct A {
    int x = 10;
    A() { this->foo(this->x); }
    void foo(int x) {std::cout << x;}
};

int main() {
    //10
    A a;
    return 0;
}
```

Initializer list

Problem

```cpp
#include <iostream>

struct A {
    const int x;
};

int main() {
// CE: the constant member
// must be initialized
    A a;
    return 0;
}
```

Solution

```cpp
#include <iostream>

struct A {
    A(int x_) : x(x_) {}
    const int x;
};

int main() {
    A a(10);
    return 0;
}
```

explicit

Problem

```cpp
#include <iostream>

class Graph {
    public:
    Graph(int n_) :
        n(n_) {}

    private:
    int n;
};

int main() {
    Graph a = 1 + 3 + 4;
    return 0;
}
```

Solution

```cpp
#include <iostream>

class Graph {
    public:
    explicit Graph(int n_) :
        n(n_) {}

    private:
    int n;
};

int main() {
    // CE
    Graph a = 1 + 3 + 4;
    return 0;
}
```

Destructor

- It is called at the end of the object's *lifetime*
- It is needed to free system resources.

Destructor

```cpp
#include <iostream>

class A {
    public:
        A() { x = new int[10]; }
        ~A() {
        std::cout << "~A";
        delete x;
    }
    private:
        int* x;
};

int main() {
    A* a = new A();
    delete a;
    return 0;
}
```

Const member functions

Such methods guarantees that class members will not be changed after they are called.

Const member functions

```cpp
#include <iostream>

class A {
    public:
        // CE
        void foo() const {value = 10;};
    private:
        int value;
};

int main() {
    A a;
    return 0;
}
```

Static variable

```cpp
#include <iostream>

void foo() {
    static int x = 0;
    x++;
    std::cout << x;
}

int main() {
    // 1
    foo();
    foo();
    // 2
    return 0;
}
```

Static member

```cpp
#include <iostream>

struct A {
    static int count;
};

int A::count = 0;

int main() {
    A a;
    a.count++;
    A b;
    // 1
    std::cout << b.count;
    return 0;
}
```

Static member function

```cpp
#include <iostream>

class A {
    public:
        static void foo() {
            count++;
            std::cout << count;
        }
    private:
        static int count;
    };

int A::count = 0;

int main() {
    A::foo();
    A::foo();
    return 0;
}
```

Static member functions and this

Static member functions **have no** this pointer

```cpp
#include <iostream>
class A {
    public:
    static void foo() {
        // CE
        std::cout << this;
    }
};

int main() {
    A a;
    a.foo();
    return 0;
}
```

Rule of three: when to use

If a class stores one of:

- pointer to a dynamic memory
- file descriptor

You need to define all three special member functions:

- copy assignment operator
- copy constructor
- destructor

in order to handle acquired resources.

Rule of three: Problem

```
struct A {
    A() { ptr = new int(1); };
    ~A() { delete ptr; }

    int* ptr;
};

int main() {
    A a;
    A b = a;
    // double free
    return 0;
}
```

Rule of three: Solution

```
struct A {
    A() { ptr = new int(1); };
    A(const A& val) {
        ptr = new int(*val.ptr);
    }
    A& operator=(const A& right) {
        if(this == &right) { return *this; }
        ptr = new int(*right.ptr);
    }
    ~A() { delete ptr; }
    int* ptr;
};

int main() {
    A a;
    A b = a;
    // no double free
    return 0;
}
```

Operator overloading

Binary

In order to maintain symmetry

- We implement it as non-member function
- We declare it as a friend

+ : as non-member

```cpp
#include <iostream>

struct A {
    A& operator +=(const A& right) {
        x += right.x;
        return *this;
    }
    friend A operator+(const A& left, const A& right);

    int x;
};

 A operator+(A& left, const A& right) {
    left += right;
    return left;
}
```

+ : as member function

```cpp
#include <iostream>

struct A {
    A& operator += (const A& right) {
        x += right.x;
        return *this;
    }
    A operator+(const A& right) {
    }

    int x;
};
```

Comparisons

```
bool operator< (const A& lhs, const A& rhs) {
    // user-defined comparison strategy
}
bool operator> (const A& lhs, const A& rhs) {
    return rhs < lhs;
}
bool operator<=(const A& lhs, const A& rhs) {
    return !(lhs > rhs);
}
bool operator>=(const A& lhs, const A& rhs) {
    return !(lhs < rhs);
}
```

Comparison

```
bool operator==(const A& lhs, const A& rhs) {
    // user-defined comparison strategy
}
bool operator!=(const A& lhs, const A& rhs) {
    return !(lhs == rhs);
}
```

Increment / decrement

```
struct X {
    X& operator++() {
        // implementation
        return *this;
    }
    X operator++(int) {
        X tmp(*this);
        operator++();
        return tmp;
    }
};
```

Function call

Functor

Class or struct that overloads function call operator ()

```cpp
#include <iostream>

struct PrintNum {
    void operator()(int x) {
        std::cout << x;
    }
};

int main() {
    PrintNum x;
    x(10);
    return 0;
}
```

Function call

Comparator

A functor that defines the comparison logic inside the operator ().

Вызов функции

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

class NaiveCompare {
    public:
        bool operator()(int x, int y) {
            return x < y;
        }
};

int main() {
    NaiveCompare a;
    std::vector<int> v = {0, 5, 4};
    std::sort(v.begin(), v.end(), NaiveCompare() );
    for (auto x : v) {
        std::cout << x;
    }
    return 0;
}
```

Restrictions

- The following operators cannot be overloaded:
  - "::" (scope resolution)
  - "." (member access)
  - ".*" (member access through pointer to member)
  - "?:"
- Operator precedence and arity cannot be changed

Inheritance

Inheritance declaration

```
#include <iostream>

class Base {
    public:
        void foo() {
            std::cout << "Base";
        }
};

class Derived : public Base {
};

int main() {
    Derived d;
    d.foo();
    return 0;
}
```

Pass by reference to Base

```
#include <iostream>

struct Base {
    void bar() {
        std::cout << "Base";
    }
};

struct Derived : Base {
};

void foo(const Base& val) {
    val.bar();
}

int main() {
    foo(Derived());
    return 0;
}
```

Default inheritance access specifier

Private

```cpp
#include <iostream>

class Base {
    public:
    void foo() {
        std::cout << "Base";
    }
};

class Derived : Base {
};

int main() {
    Derived d;
    // CE
    d.foo();
    return 0;
}
```

Public

```cpp
#include <iostream>

struct Base {
    void foo() {
        std::cout << "Base";
    }
};

struct Derived : Base {
};

int main() {
    Derived d;


    d.foo();
    return 0;
}
```

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Figure: Specifiers relationship

Not accessible

```cpp
#include <iostream>

class Base {
    void foo() {
        std::cout << "Base";
    }
};

class Derived : public Base {
};

int main() {
    Derived d;
    d.foo();
    return 0;
}
```

Protected

Allows to propagate class members accessibility down the hierarchy preserving inaccessibility from the outside.

Protected

```cpp
#include <iostream>

class A {
    protected:
        int x = 10;
};

class B : private A {
    // x -> private
};

// x is inaccessible inside C
class C : public B {
    public:
    void foo() {
        std::cout << x;
    }
};
```

```cpp
#include <iostream>

class A {
    protected:
        int x = 10;
};

class B : public A {
    // x -> protected
};

// x is accessible inside C
class C : private B {
    public:
    void foo() {
        std::cout << x;
    }
};
```

Protected

```cpp
#include <iostream>

class A {
    protected:
        int x = 10;
};

class B : protected A {
    // x -> protected
};

// x is accessible inside C
class C : private B {
    public:
    void foo() {
        std::cout << x;
    }
};
```

Name hiding

```
struct Base {
    void foo() {
        std::cout << "Base";
    }
};

struct Derived : Base {
    void foo() {
        std::cout << "Derived";
    }
};

int main() {
    Derived d;
    // Derived
    d.foo();
    return 0;
}
```

Name hiding: access Base

```cpp
struct Base {
    void foo() {
        std::cout << "Base";
    }
};

struct Derived : Base {
    public:
    void foo() {
        Base::foo();
        std::cout << "Derived";
    }
};

int main() {
    Derived d;
    d.foo(); // Base Derived
    return 0;
}
```

Name hiding: access Base

Problem

```cpp
struct A {
    int x = 10;
};

struct B {
    int x = 20;
};

struct C : A, B {
    C() {
        std::cout << x; // CE
    }
};

int main() {
    C c;
    return 0;
}
```

Solution

```cpp
struct A {
    int x = 10;
};

struct B {
    int x = 20;
};

struct C : A, B {
    C() {
        std::cout << A::x;
    }
};

int main() {
    C c;
    return 0;
}
```

using

```cpp
#include <iostream>

struct B {
    void g(char x) { std::cout << x; }
};

struct D : B {
    using B::g;
    void g(int x) { std::cout << x; }
};

int main() {
    D d;
    d.g(1); // 1
    d.g('a'); // a
    return 0;
}
```

Constructor and destructor: Invocation order

Constructor

From left to right and from top to bottom (see the picture below)

destructor

In the reverse order of the order in which Constructor are called

Constructor and destructor

```cpp
struct A {
    A() { std::cout << "A"; }
    ~A() { std::cout << "~A"; }
};

struct B {
    B() { std::cout << "B"; }
    ~B() { std::cout << "~B"; }
};

struct C : A, B {
    C() { std::cout << "C"; }
    ~C() { std::cout << "~C"; }
};
int main() {
    C c;
    return 0;
}
```

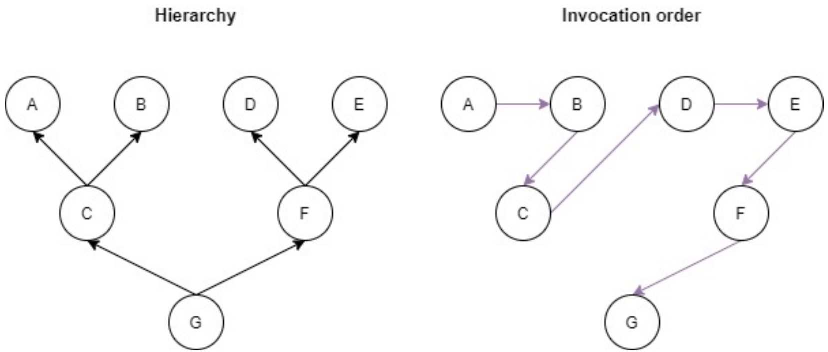

Figure: Constructor invocation

Dimond

```
#include <iostream>

struct A {
    int x;
};

struct B : A {};

struct C : A {};

struct D : B, C {};

int main() {
    D d;
    d.x; // CE
    return 0;
}
```

Name resolution using operator ::

```
#include <iostream>

struct A {
    int x;
};

struct B : A {};

struct C : A {};

struct D : B, C {};

int main() {
    D d;
    d.B::x;
    return 0;
}
```