四川大學
SICHUAN UNIVERSITY

Object Oriented Programming—C++
Lecture7 Classes (II)

Qijun Zhao
College of Computer Science
Sichuan University

Spring 2023

Contents

1.1 class vs. struct

C/C++ Structure

A structure (struct) is a collection of variables of the same or different data types under a single name

C++ Class

A class (class) extends the concept of structure to hold functions as members

struct vs. class

Structures and classes are semantically equivalent. In general, struct represents passive objects, while class active objects

1.2 Class Members - Data and Function Members

Data Member

Data within a class are called data members or class fields

Function Member

Functions within a class are called function members or methods

1.3.1 RAII Idiom - Resource Acquisition is Initialization

Holding a resource is a class invariant, and is tied to object lifetime

RAII Idiom consists in three steps:

- Encapsulate a resource into a class (constructor)
- Use the resource via a local instance of the class
- The resource is automatically released when the object gets out of scope *(destructor)*

Implication 1: C++ programming language does not require the garbage collector!!

Implication 2: The programmer has the responsibility to manage the resources

struct/class Declaration and Definition

struct declaration and definition

```
struct A;       // struct declaration

struct A {      // struct definition
    int x;      // data member
    void f();   // function member
};
```

class declaration and definition

```
class A;        // class declaration

class A {       // class definition
    int x;      // data member
    void f();   // function member
};
```

struct/class Function Declaration and Definition

```
struct A {
    void g();          // function member declaration

    void f() {         // function member declaration
        cout << "f"; // inline definition
    }
};

void A::g() {          // function member definition
    cout << "g";     // out-of-line definition
}
```

struct/class Members

```cpp
struct B {
    void g() { cout << "g"; } // function member
};

struct A {
    int  x;                   // data member
    B    b;                   // data member
    void f() { cout << "f"; } // function member
};

A a;
a.x;
a.f();
a.b.g();
```

1.3.2 Smart pointers

Smart pointer is a pointer-like type with some additional functionality, e.g. automatic memory deallocation (when the pointer is no longer in use, the memory it points to is deallocated), reference counting, etc.

C++11 provides three smart pointer types:

- **std::unique ptr**
- **std::shared ptr**
- **std::weak ptr**

Smart pointers prevent most situations of memory leaks by making the memory deallocation automatic

Smart Pointers Benefits

- If a smart pointer goes out-of-scope, the appropriate method to release resources is called automatically. The memory is not left dangling

- Smart pointers will automatically be set to `nullptr` if not initialized or when memory has been released

- `std::shared_ptr` provides automatic reference count

- If a special `delete` function needs to be called, it will be specified in the pointer type and declaration, and will automatically be called on delete

std::unique ptr - Unique Pointer

std::unique_ptr is used to manage any dynamically allocated object that is <u>not shared</u> by multiple objects

```
#include <iostream>
#include <memory>
struct A {
    A()  { std::cout << "Constructor\n"; } // called when A()
    ~A() { std::cout << "Destructor\n";  } // called when u_ptr1,
};                                          // u_ptr2 are out-of-scope
int main() {
    auto raw_ptr = new A();
    std::unique_ptr<A> u_ptr1(new A());
    std::unique_ptr<A> u_ptr2(raw_ptr);
//  std::unique_ptr<A> u_ptr3(raw_ptr); // no compile error, but wrong!!
                                        // (same pointer)
//  u_ptr1 = &raw_ptr;  // compile error (unique pointer)
//  u_ptr1 = u_ptr2;    // compile error (unique pointer)
    u_ptr1 = std::move(u_ptr2); // delete u_ptr1;
}                               // u_ptr1 = u_ptr2;
                                // u_ptr2 = nullptr
```

std::unique ptr - Unique Pointer

std::unique_ptr methods

- get() returns the underlying pointer
- operator* operator-> dereferences pointer to the managed object
- operator[] provides indexed access to the stored array (if it supports random access iterator)
- release() returns a pointer to the managed object and releases the ownership
- reset(ptr) replaces the managed object with ptr

Utility method: std::make unique<T>() creates a unique pointer of a class T that manages a new object

std::unique ptr - Unique Pointer

```cpp
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    std::unique_ptr<A> u_ptr1(new A());
    u_ptr1->value;      // dereferencing
    (*u_ptr1).value;    // dereferencing

    auto u_ptr2 = std::make_unique<A>();  // create a new unique pointer

    u_ptr1.reset(new A());              // reset
    auto raw_ptr = u_ptr1.release();    // release
    delete[] raw_ptr;

    std::unique_ptr<A[]> u_ptr3(new A[10]);
    auto& obj = u_ptr3[3];              // access
}
```

std::unique ptr - Unique Pointer

Implement a custom deleter

```cpp
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    auto DeleteLambda = [](A* x) {
        std::cout << "delete" << std::endl;
        delete x;
    };

    std::unique_ptr<A, decltype(DeleteLambda)>
        x(new A(), DeleteLambda);
} // print "delete"
```

std::shared ptr - Shared Pointer

std::shared ptr is the pointer type to be used for memory that can be owned by multiple resources at one time

std::shared ptr maintains a reference count of pointer objects. Data managed by

std::shared ptr is only freed when there are no remaining objects pointing to the data

```cpp
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    std::shared_ptr<A> sh_ptr1(new A());
    std::shared_ptr<A> sh_ptr2(sh_ptr1);
    std::shared_ptr<A> sh_ptr3(new A());
    sh_ptr3 = nullptr; // allowed, the underlying pointer is deallocated
                       //   sh_ptr3 : zero references
    sh_ptr2 = sh_ptr1; // allowed // sh_ptr1, sh_ptr2: two references
    sh_ptr2 = std::move(sh_ptr1); // allowed // sh_ptr1: zero references
}                                            // sh_ptr2: one references
```

std::shared ptr - Shared Pointer

std::shared ptr methods

- get() returns the underlying pointer
- operator* operator-> dereferences pointer to the managed object
- use count() returns the number of objects referring to the same managed object
- reset(ptr) replaces the managed object with ptr

Utility method: std::make shared() creates a shared pointer that manages a new object

std::shared ptr - Shared Pointer

```cpp
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    std::shared_ptr<A> sh_ptr1(new A());
    auto sh_ptr2 = std::make_shared<A>();  // std::make_shared
    std::cout << sh_ptr1.use_count();  // print 1

    sh_ptr1 = sh_ptr2;                 // copy
//  std::shared_ptr<A> sh_ptr2(sh_ptr1); // copy (constructor)
    std::cout << sh_ptr1.use_count();  // print 2
    std::cout << sh_ptr2.use_count();  // print 2

    auto raw_ptr = sh_ptr1.get();      // get
    sh_ptr1.reset(new A());            // reset
    (*sh_ptr1).value = 3;              // dereferencing
    sh_ptr1->value = 2;                // dereferencing
}
```

std::weak ptr - Weak Pointer

A std::weak ptr is simply a std::shared ptr that is allowed to dangle (pointer not deallocated)

```
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    auto ptr = new A();
    std::weak_ptr<A>   w_ptr(ptr);
    std::shraed_ptr<A> sh_ptr(new A());

    sh_ptr = nullptr;
//  delete sh_ptr.get(); // double free or corruption

    w_ptr = nullptr;
    delete w_ptr; // ok valid
}
```

std::weak ptr - Weak Pointer

It must be converted to std::shared ptr in order to access the referenced object

std::weak ptr methods

- use count() returns the number of objects referring to the same managed object
- reset(ptr) replaces the managed object with ptr
- expired() checks whether the referenced object was already deleted (true, false)
- lock() creates a std::shared ptr that manages the referenced object

std::weak ptr - Weak Pointer

```
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    auto sh_ptr1 = std::make_shared<A>();
    std::cout << sh_ptr1.use_count();  // print 1
    std::weak_ptr<A> w_ptr = sh_ptr1;
    std::cout << w_ptr.use_count();    // print 1

    auto sh_ptr2 = w_ptr.lock();
    std::cout << kk.use_count();       // print 2 (sh_ptr1 + sh_ptr2)

    sh_ptr1 = nullptr;
    std::cout << w_ptr.expired();     // print false
    sh_ptr2 = nullptr;
    std::cout << w_ptr.expired();      // print true
}
```

2. Class Hierarchy

Child/Derived Class or Subclass

A new class that inherits variables and functions from another class is called a derived or child class

Parent/Base Class

The closest class providing variables and functions of a derived class is called parent or base class

Extend a base class refers to creating a new class which retains characteristics of the base class and on top it can add (and never remove) its own members

Class Hierarchy

```
struct A {          // base class
    int value = 3;
    void g() {}
};

struct B : A {      // B is a derived class of A (B extends A)
    int data = 4;   // B inherits from A
    int f() { return data; }
};

A a;
B b;
a.value;
b.g();
```

Class Hierarchy

```
struct A {};
struct B : A {};

void f(A a) {}       // copy
void g(B b) {}       // copy

void f_ref(A& a) {} // the same for A*
void g_ref(B& b) {} // the same for B*

A a;
B b;
f(a); // ok, also f(b), f_ref(a), g_ref(b)
g(b); // ok, also g_ref(b), but not g(a), g_ref(a)

A a1 = b;     // ok, also  A& a2 = b
// B b1 = a; // compile error
```

2.1 Access specifiers

The **access specifiers** define the visibility of inherited members of the subsequent base class. The keywords `public` , `private` , and `protected` specify the sections of visibility

The goal of the access specifiers is to prevent a direct access to the internal representation of the class for avoiding wrong usage and potential inconsistency (access control)

- public: No restriction (function members, derived classes, outside the class)
- protected: Function members and derived classes access
- private: Function members only access (internal)

`struct` has default `public` members

`class` has default `private` members

Access specifiers

```
struct A1 {
    int value;   // public (by default)
protected:
    void f1() {} // protected
private:
    void f2() {} // private
};

class A2 {
    int data;    // private (by default)
};
struct B : A1 {
   void h1() { f1(); } // ok, "f1" is visible in B
// void h2() { f2(); } // compile error "f2" is private in A1
};

A1 a;
a.value;  // ok
// a.f1() // compile error protected
// a.f2() // compile error private
```

2.2 Inheritance Access Specifiers

The access specifiers are also used for defining how the visibility is propagated from the base class to a specific derived class in the inheritance

Member declaration		Inheritance		Derived classes
public protected private	→	**public**	→	public protected \
public protected private	→	**protected**	→	protected protected \
public protected private	→	**private**	→	private private \

Inheritance Access Specifiers

```cpp
struct A {
    int var1; // public
protected:
    int var2; // protected
};

struct B : protected A {
    int var3; // public
};

B b;
// b.var1; // compile error, var1 is protected in B
// b.var2; // compile error, var2 is protected in B
b.var3;    // ok, var3 is public in B
```

Inheritance Access Specifiers

```
class A {
public:
    int var1;
protected:
    int var2;
};

class B1 : A {};        // private inheritance

class B2 : public A {}; // public inheritance

B1 b1;
// b1.var1; // compile error, var1 is private in B1
// b1.var2; // compile error, var2 is private in B1

B2 b2;
b2.var1;    // ok, var1 is public in B2
```

3. Class Constructor and Destructor

3.1 Intro

Constructor [ctor]

A constructor is a special member function of a class that is executed when a new instance of that class is created

Goals: initialization and resource acquisition

Syntax: T(...) same named of the class and no return type

- A constructor is supposed to initialize all data members
- We can define multiple constructors with different signatures
- Any constructor can be constexpr

3.2 Default Constructor

Default Constructor

The default constructor T() is a constructor with no argument

Every class has always either an implicit or explicit default constructor

```
struct A {
    A()    {} // explicit default constructor
    A(int) {} // user-defined (non-default) constructor
};
```

```
struct A {
    int x = 3; // implicit default constructor
};
A a{}; // ok
```

- An implicit default constructor is constexpr

Default Constructor Examples

```
struct A {
    A() { cout << "A"; }  // default constructor
};

A  a1;                  // call the default constructor
// A a2();              // interpreted as a function declaration!!
A a3{};                 // ok, call the default constructor
                        // direct-list initialization (C++11)

A  array[3];            // print "AAA"

A* ptr = new A[4]; // print "AAAA"
```

Deleted Default Constructor

The implicit default constructor of a class is marked as deleted if (simplified):

- It has any user-defined constructor

```
struct A {
    A(int x) {}
};
// A a; // compile error
```

- It has a non-static member/base class of reference/const type

```
struct NoDefault { // deleted default constructor
    int&      x;
    const int y;
};
```

Deleted Default Constructor

- It has a non-static member/base class which has a deleted (or inaccessible) default constructor

```
struct A {
    NoDefault var;         // deleted default constructor
};
struct B : NoDefault {}; // deleted default constructor
```

- It has a non-static member/base class with a deleted or inaccessible destructor

```
struct A {
private:
    ~A() {}
};
```

3.3 Class Initialization

3.3.1 Initializer List

The Initializer list is used for initializing the data members of a class or explicitly call the base class constructor before entering in the constructor body

(Not to be confused with std::initializer list)

```
struct A {
    int x, y;

    A(int x1) : x(x1) {}  // ": x(x1)" is the Initializer list
                          // direct initialization syntax

    A(int x1, int y1) :   // ": x{x1}, y{y1}"
        x{x1},            // is the Initializer list
        y{y1} {}          // direct-list initialization syntax
};                        // (C++11)
```

3.3.2 In-Class Member Initializer

C++11 In-class non-static data members can be initialized where they are declared (NSDMI). A constructor can be used when run-time initialization is needed

```
struct A {
    int         x   = 0;       // in-class member initializer
    const char* str = nullptr; // in-class member initializer

    A() {} // "x" and "str" are well-defined if
           // the default constructor is called

    A(const char* str1) : str{str1} {}
};
```

3.3.3 Initialization Order

Class members initialization follows the order of declarations and not the order in the initialization list

```
struct ArrayWrapper {
    int* array;
    int  size;

    A(int user_size) :
        size{user_size},
        array{new int[size]} {}
        // wrong!!: "size" is still undefined
};

ArrayWrapper a(10);
cout << a.array[4]; // segmentation fault
```

3.3.4 Uniform Initialization

Uniform Initialization (C++11)

Uniform Initialization {}, also called *list-initialization*, is a way to fully initialize any object independently from its data type

- **Minimizing Redundant Typenames**
  - In function arguments
  - In function returns
- Solving the **"Most Vexing Parse" problem**
  - Constructor interpreted as function prototype

3.4 Constructors and Inheritance

Class constructors are <u>never</u> inherited

A *Derived* class <u>must</u> call *implicitly* or *explicitly* a *Base* constructor <u>before</u> the current class constructor

Class constructors are called <u>in order</u> from the top Base class to the most Derived class (C++ objects are constructed like onions)

```
struct A {
    A() { cout << "A" };
};
struct B1 : A { // call "A()" implicitly
    int y = 3;  // then, "y = 3"
};
struct B2 : A { // call "A()" explicitly
    B2() : A() { cout << "B"; }
};
B1 b1; // print "A"
B2 b2; // print "A", then print "B"
```

3.5 Delegate Constructor

The problem:

Most constructors usually perform identical initialization steps before executing individual operations

C++11 A **delegate constructor** calls another constructor of the same class to reduce the repetitive code by adding a function that does all of the initialization steps

```
struct A {
    int   a;
    float b;
    bool  c;
    // standard constructor:
    A(int a1, float b1, bool c1) : a(a1), b(b1), c(c1) {
        // do a lot of work
    }

    A(int a1, float b1) : A(a1, b1, false)  {} // delegate construtor
    A(float b1)         : A(100, b1, false) {} // delegate construtor
};
```

3.6 explicit Keyword

explicit

The explicit keyword specifies that a *constructor* or *conversion function* (C++11) does not allow implicit conversions or copy-initialization

```cpp
struct A {
    A(int) {}
    A(int, int) {}
};


struct B {
    explicit B(int) {}
    explicit B(int, int) {}
};
```

```cpp
A a1(2);           // ok
A a2 = 1;          // ok (implicit)
A a3{4, 5};        // ok. Selected A(int, int)
A a4 = {4, 5};     // ok. Selected A(int, int)


B b1(2);              // ok
// B b2 = 1;          // error implicit conversion
B b3{4, 5};           // ok. Selected B(int, int)
// B b4 = {4, 5};     // error implicit conversion
B b5 = (B) 1;         // OK: explicit cast
```

3.7 Copy Constructor

Copy Constructor

A copy constructor T(const T&) creates a new object as a *deep copy* of an existing object

```
struct A {
    A()          {} // default constructor
    A(int)       {} // non-default constructor
    A(const A&) {} // copy constructor
}
```

- Every class always defines an implicit or explicit copy constructor
- Even the copy constructor implicitly calls the default Base class constructor
- Even the copy constructor is considered a non-default constructor

Copy Constructor Example

```
struct Array {
    int  size;
    int* array;

    Array(int size1) : size{size1} {
        array = new int[size];
    }
    // copy constructor, ": size{obj.size}" initializer list
    Array(const Array& obj) : size{obj.size} {
        array = new int[size];
        for (int i = 0; i < size; i++)
            array[i] = obj.array[i];
    }
};
Array x{100}; // do something with x.array ...
Array y{x};   // call "Array::Array(const Array&)"
```

Copy Constructor Usage

The copy constructor is used to:

- Initialize one object from another one having the same type
  - Direct constructor
  - Assignment operator

```
A a1;
A a2(a1);     // Direct copy initialization
A a3{a1};     // Direct copy initialization
A a4 = a1;    // Copy initialization
A a5 = {a1}; // Copy list initialization
```

- Copy an object which is passed by-value as input parameter of a function

```
void f(A a);
```

- Copy an object which is returned as result from a function

```
A f() { return A(3); } // * see RVO optimization
```

Copy Constructor Usage Examples

```
struct A {
    A() {}
    A(const A& obj) { cout << "copy"; }
};

void f(A a) {} // pass by-value

A g() { return A(); };

A a;
A b = a;      // copy constructor (assignment)    "copy"
A c(b);       // copy constructor (direct)        "copy"
f(b);         // copy constructor (argument)      "copy"
g();          // copy constructor (return value)  "copy"
A d = g();    // * see RVO optimization          (depends)
```

Pass by-value and Copy Constructor

```
struct A {
    A() {}
    A(const A& obj) { cout << "expensive copy"; }
};

struct B : A {
    B() {}
    B(const B& obj) { cout << "cheap copy"; }
};

void f1(B b) {}
void f2(A a) {}

B b1;
f1(b1); // cheap copy
f2(b1); // expensive copy!! It calls A(const A&) implicitly
```

3.8 Move Constructor

Is copying enough?

We’ ve learned about the default constructor, destructor, and the copy constructor and assignment operator.

- We can create an object, get rid of it, and copy its values to another object!
- Is this ever insufficient?

This can be wasteful!

Let's say we had to copy our current StringTable into another, whose reference is given to us, and we have no use for our StringTable afterwards.

```
class StringTable {
  public:
    StringTable() {}
    StringTable(const StringTable& st) {}
    // functions for insertion, erasure, lookup, etc.,
    // but no move/dtor functionality
    // ...

  private:
    std::map<int, std::string> values;
}
```

The copy constructor will copy every value in the values map one by one! Very slowly!

There are six special member functions!

These functions are generated only when they're called (and before any are explicitly defined by you):

```
class Widget {
  public:
    Widget();                                   // default constructor
    Widget (const Widget& w);                   // copy
    Widget& operator = (const Widget& w);       // copy assignment operator
    ~Widget();                                  // destructor
    Widget (Widget&& rhs);                      // move constructor
    Widget& operator = (Widget&& rhs);          // move assignment operator
}
```

Let’ s talk about it now!

Move operations

- Move constructors and move assignment operators will perform "memberwise moves."

Unlike copy operations!

- Defining a move assignment operator **prevents generation** of a move copy constructor, and vice versa.
    - If the move assignment operator needs to be re-implemented, there'd likely be a problem with the move constructor!

Caveats

Move constructors and operators are only generated if:

- No copy operations are declared
- No move operations are declared
- No destructor is declared

Declaring any of these will get rid of the default C++ generated operations.

Caveats

If we want to explicitly support move operations, we can set the operators to default:

???

```
Widget(Widget&&) = default;
Widget& operator=(Widget&&) = default;          // support moving

Widget(const Widget&) = default;
Widget& operator=(const Widget&) = default;     // support copying
```

四川大學
SICHUAN UNIVERSITY

Coding for love, Coding for the world

Qijun Zhao
qjzhao@scu.edu.cn