OOP Final Review

OOP Characteristics

- Everything is an object
- A program is a bunch of objects telling each other what to do by sending messages
- Each objects has its own memory made up of other objects
- Every object has a type
- All objects of a particular type can receive the same messages

C++ Basics

Const 类型

```
const int x = 123;
x = 27; // illegal

int y = x; // ok, copy const to non-const
const int z = y; // ok
```

Compile time and Run-time constants

- Compile time constants `const int bufsize = 1024`
  - value must be initialized
- Run-time constants

```
const int class_size = 12;
int final[class_size]; //ok

int x;
cin >> x;
const int size = x;
double class[size]; // error
```

const 和指针 Pointer

- 常量指针：`char* const p = "abc"`。不能赋予这个指针新的地址，相当于地址是 const 类型，但是 p 指向的值可以改变。(p 是常数)
- `const char* p = 'abc'` 这种情况下 `p` 指向的是一个 `const char` 类型的值，因此指向的值不能改变。 (*p 是常数)

```
char * const q = 'abc';
*q = 'c';//ok
q++;//Error

const char* p = 'abc';
*p = 'c'; //error
```

不能把 const 类型的变量赋值给对应的指针，因为可能会带来 const 变量的改变。

可以把非 const 类型的值赋给对应的 const 型变量。

```
int i;
const int ci = 3;
int *ip;

ip = &i;
ip = &ci; // Error ip 是一个 int* 类型的指针，而 ci 是一个 const int 类型的变量。不能将 const int* 类型的指针赋值给 int* 类型的指针。

const int *cip;
cip = &ci; //ok
```

- 通过const 引用，我们无法修改它所引用的对象

```
int value = 10;
const int &ref = value;
ref = 20;// Error
```

- 在类中，const 可以用来修饰成员函数，表示这个函数不能修改对象的成员变量。

```
class MyClass {
    public:
        int getvalue() const { //该函数不能修改 value的值
            return value;
        }
    private:
        int value;
}
```

- 使用 const 关键字实例化类对象后，被实例化的类对象只能调用类中的常量成员函数，且被初始化后不可被修改，常量成员函数可以被重载，普通实例化对象优先调用普通成员函数，初始化后可随意修改。

```
class Test {
public:
    Test(int id_, int age_) {
        this->id = id_;
```

```
        this->age = age_;
    }

    void show() {
        std::cout << this->id << std::endl;
        std::cout << this->age << std::endl;
    }

    void show() const {
        std::cout << this->id << std::endl;
        std::cout << this->age << std::endl;
    }
public:
    int id;
    int age;
};

void test()
{
    const Test test01(123, 345);
    test01.id=666;//错误，被初始化后不能再修改
    test01.show();//调用且只能调用常量成员函数


    Test test02(123,456);
    test02.id = 111;//可以随意修改
    test02.show();//优先调用普通成员函数
}
```

inline

- 功能：将函数展开为语句，类似宏的字符串替换，但更安全
- 效果：导致代码长度增加，运行时间减少
- 在类内定义的函数，默认当作有 `inline` 关键字的函数进行处理
- 关键字 **`inline`** 必须与函数定义体放在一起才能使函数成为内联，仅将 **`inline`** 放在函数声明前面不起任何作用。

引用

- 必须跟对象绑定。
- 一旦绑定，无法再次绑定其它对象
- 不可为空

```
char c; // a character
char *p = &c; // a pointer to a character
char &r = c; // a reference to a character


int& y = x;
y = z; // change value of x to value of z
```

Type restrictions

- No references to references
- No pointers to references, but reference to pointer is ok.

```
int& *p; //illegal
void f(int*& p); //ok
```

- No arrays of references

Static

- 被用来控制变量的存储方式和作用范围。
- 在函数中声明变量时，static 关键字指定变量只初始化一次，并在之后调用该函数时保留其状态。
- static 在修饰局部变量时，会改变局部变量的存储位置，从而使得局部变量的生命周期变长。
- static 关键字没有赋值时，默认为 0
- 被 static 修饰的全局变量只能在自己的源文件中使用

C++中的static

- 声明为static的类成员称为类的静态成员，分为如下两类：
  - 用static修饰的成员变量，称之为**静态成员变量**
  - 用static修饰的成员函数，称之为**静态成员函数**
  - 静态的成员变量一定要在类外进行初始化
- 静态成员为所有类对象所共享，不属于某个具体的实例
- 静态成员变量必须在类外定义，定义时不添加 static 关键字

```
class A{
   private:
    static int n;
};
int A::n = 0;
```

- 静态成员函数没有隐藏的this指针，不能访问任何非静态成员。

Class

Objects = Attributes + Services

- Data: the properties, or status
- Operations: the functions

> Objects: represent things, events, respond to messages at run-time
>
> Classes: define properties of instances

- `::resolver`
  - `<Class Name>:: <function name>`
  - `::<function name>`
- `#include`
  - `#include "xx.h"` : usually search in the current directory, implementation defined
  - `#include <xx.h>` : search in the specified directories, depending on your development environment.
- Encapsulation
  - Bundle data and methods together
  - Hide the details of dealing with the data
  - Restrict access only to the publicized methods.
- Initializer list
  - Order of initialization is order of declaration, not the order in the initializer list
  - Destroyed in the reverse order

```
class Point {
private:
 const float x, y;
public:
 Point(float xa, float ya)
 : y(ya), x(xa) {}
}

//Aggregate initialization

struct X {
    int i; float f; char c;
};

X x2[3] = {{1, 1,1, 'a'}, {2, 2,2, 'b'}};
```

- Constant members in class has to be initialized in the ctor's initializer list.
- 构造函数
  - 不能是虚函数
  - 不能有返回值
  - 初始化列表中的初始化顺序和变量的声明顺序一致

- 析构函数
  - 一般写成虚函数

```
virtual ~A() {}; //不能有参数
```

  - `new/delete`
    - new 时调用构造函数，delete 的时候调用析构函数
    - delete 只是释放内存空间，指针的值没有变

```
A *p = new A[10];
delete p; // 只调用 p 所指的对象的析构
delete [] p; //调用数组每个元素的析构函数，然后释放整个数组的内存
```

Composition & Inheritance

> Composition: construct new object with existing objects (has-a)
>
> Inheritance: clone an existing class and extend it. (is-a)

Composition

```
class Person { … };
class Currency { … };
class SavingsAccount {
public:
 SavingsAccount(
     const string& name,
     const string& address,
     int cents);
 ~SavingsAccount();
 void print();
private:
 Person m_saver;
 Currency m_balance;
};

SavingAccount::SavingAccount(const string& name, const string& address, int
cents) : m_saver(name, address), m_balance(0, cents) {}
```

Inheritance

Base class is always constructed first.

If no explicit arguments are passed to base class, the default constructor will be called.

Destructors are called in exactly the reverse order of the constructors.

- Name hiding
  - If you redefine a member function in the derived class, all the other overloaded functions in the base class are inaccessible.
- Friends
  - To expilicitly grant access to a function that isn't a member fo the structure
  - The class itself controls which code has access to its members
  - Can declare a global function, a member function of another class, or even an entire class, as a friend.

没有被继承的成员

- 构造函数没有被继承，但父类的构造会被自动调用。析构同理
- 赋值运算符不会被继承
- 友元不会被继承

继承时构造函数的调用顺序

1. 基类的构造函数（可能递归下去）
2. 派生类成员变量的构造函数
3. 派生类自身的构造函数

How inheritance affects access

- `class B : private A` public and protected members are private in B, private members are not accessible.
- `class B: protected A` public and protected members in A are protected in B, private members are not accessible.
- `class B : public A` : public members in A are public in B, protected are protected, private are not accessible.

Upcasting

Students are human beings. You are students. So you are human being

```
Manager pete("Pete", "1234", "Bakery");
Employee * ep = &pete; //Upcast
Employee & er = pete; // Upcast
```

Polymorphism

> 多态：同一句代码实际执行中可能会有多种执行方式
>
> Binding: which function to be called
>
> - Static binding: call the function as the declared type
> - Dynamic binding: call the function according to the real type of the object

- 静态链接
  - 函数的调用在程序开始运行之前就已经确定了
- 虚函数 Virtual Function

- 一种用于实现多态的机制，核心理念是通过基类访问派生类定义的函数。称为动态链接
- 用于区分派生类中和基类同名的方法函数，需要将基类的成员函数类型声明为 `virtual`
- 基类中的析构函数一定要为虚函数。
- 纯虚函数： `virtual int func() = 0` 。表明该函数没有主体，基类中没有给出有意义的实现方式，需要在派生类中进行扩展。
- override语法：派生类中可以用override关键字来声明，表示对基类虚函数的重载。

```
class Base{
    virtual void display() const {cout << x << endl;}//const
};
class Derived: public Base{
    void display() const{cout << x << "," << y << endl;}//如果没有const，不会形成多态
};
Base *p;
p→display();//多态
```

```
Ellipse elly(20f, 40f);
Circle circ(60f);
elly = circ;
```

- Area of `circ` is sliced off(被截断), only the part of `circ` fits in `elly` gets copied.
- The vptr from circ is ignored. vptr in `ellly` still points to the Ellipse vtable.

```
Ellipse *elly = new Ellipse();
Circle *circ = new Circle();
elly = circ;//此时是向上造型， elly and circ point to the same Circle object
```

Return types relaxation: Applies to pointer and reference types

```
class Expr {
public:
 virtual Expr* newExpr();
 virtual Expr& clone();
 virtual Expr self();
};
class BinaryExpr : public Expr {
public:
 virtual BinaryExpr* newExpr(); // ok
 virtual BinaryExpr& clone(); // ok
 virtual BinaryExpr self(); // Error!
};
```

- 虚函数需要借助指针和引用达到多态的效果
  - 如果基类指针/引用指向基类，那就调用基类的相关成员函数

        - 如果基类指针指向派生类，则调用派生类的成员函数
        - 派生类指针不能指向基类
    - 注意事项
        - 静态成员函数不能是虚函数，因为 static 成员函数不属于任何对象
        - 构造函数不能是虚函数
        - 析构函数推荐写成虚函数（不然 delete 基类指针时，无法调用派生类的析构函数）
- 对于指针和引用
    - 不是虚函数的时候，调用的函数取决于指针和引用的变量类型
    - 是虚函数的时候，调用函数取决于指针和引用指向的变量类型（指向基类调用基类，指向派生类调用派生类）
    - 如果派生类中没有新的同名函数，那么执行的都是基类里的

Copy Constructor

- Copying: create a new object from an existing one.
    - Copying is implemented by the copy constructor.
    - Has the unique signature `T::T(const T&)`
        - call by reference is used for the explicit argument
    - C++ builds a copy ctor for you if you don't provide one.
        - Copies each member variable（如果被拷贝的那个对象有一个成员是个 Object，那么调用该 object 的拷贝函数）
        - Copies each pointer （如果有成员变量是指针，会和原来对象一样指向同一块内存。如果有一个对象被析构，那么这块内存就被 delete，这就变成了无效内存）
- When are copy ctors called?

  如果定义变量时直接给变量用同类型的变量赋值，调用的就是拷贝构造函数，如果是定义之后再赋值，就是调用了重载之后的等号。
    - During initialization

```
Person baby_a("Fred");
Person baby_b = baby_a; // not an assignment
Person baby_c ( baby_a ); // not an assignment
```

    - During call by value

```
void roster(Person) {

}
Person child("Ruby"); //create object
roster(child); //call function
```

    - During function return

```
Person captain() {
    Person player("George");
    return player;
}
Person who = captain();
```

- Construction vs. assignment
  - Every object is constructed once
  - Every object should be destroyed once
  - Once an object is constructed, it can be the target of many assignment operations
- 浅拷贝：编译器生成的默认构造函数只拷贝每个变量的值，如果被拷贝的是指针，拷贝的指针和原来的指针指向同一块内存，原来的指针所指向的内存释放了或被修改会对拷贝的对象造成影响。
- 深拷贝：手动定义拷贝构造函数，手动分配内存，并把原指针指向的内容复制过来。
- 声明时 `=default` 代表使用默认的拷贝构造函数， `=delete` 代表禁用拷贝构造函数。
- 在声明构造函数时，可以使用 `explicit` 关键字来防止拷贝/隐式类型转化

```
class T{
    int x;
    public:
        explicit T(const T &t) {x = t.x;}
        explicit T(const int y) { x = y;}
};

int main() {
    T t1 = 1; // error
    T t1(1);//ok
    T t2 = t1;//error
    T t2(t1);//ok
}
```

Static

- Two basic meanings
  - Persistent storage: allocated once at a fixed address
  - Visibility of a name: internal linkage
- Static applied to objects
  - Construction occurs when definition is encountered.
    - Constructor called at most once
    - The constructor arguments must be satisfied
  - Global objects

```
static X global_x1(12, 34);
static X global_x2(8, 16);
```

  - Constructors are called before `main()` is entered.
    - Order controlled by appearance in file
    - `main()` is no longer the first function being called
    - Destructors called when
      - `main()` exits
      - `exit()` is called

- 静态成员为所有类对象所共享，不属于某个具体的实例
- 静态成员必须在类外定义，定义时不添加 static 关键字
- 静态成员函数没有隐藏的 this 指针，不能访问任何非静态成员

Operator Overloading

> Allows user-defined types to act like built-in types
>
> Another way to make a function call

- Operators you can't overload: `.` , `.*` , `::` , `?:` `sizeof, typeid, static_cast, dynamic_cast, const_cast, reinterpret_cast` .
- Restrictions
  - Only existing operators can be overloaded
  - Overloaded operators must (优先级和操作数个数)
    - Preserve number of operands
    - Preserve precedence
- C++ overloaded operator
  - Can be a member funcion: implicit first argument

```
String String::operator+(const String& that);
```

  - Can be a global function, explict arguments

```
String operator+(const String& l, const String& r);
```

- Operator as Member functions
  - Implicit first argument
  - Full access to class definition and all data
  - No type conversion performed on receiver

```
Integer x(1), y(5), z;
x + y ;//x.operator+(y)
z = x + 3;//ok
z = 3 + y;//error
```

  - For binary operators (`+,-,*`), the member functions requrie one argument.
  - For unary operators(`unary -, !`), the member functions require no arguments.

- Operator as global function
  - Type conversions performed on both arguments

```
class Integer {
    public:
        friend Integer operator+(const Integer&, const Integer&);
}

Integer operator+(const Interger&, const Integer&);
Interger x, y;
x + y; // operator+(x,y)
z = 3 + y; // operator+(Integer(3), y)
z  = 3 + 7; //Integer(10)
```

- The prototypes of operators
  - +-*/%^&|~: `T operator X(const T& l, const T& r)`
  - ! && || << == : `bool operator X(const T& l, const T& r)`
  - []: `E& T::operator [](int index)`
- Operator ++ and --
  - Postfix forms take an int argument: compiler will pass in `0` as that argument

```
class Interger{
public:
    Integer& operator++();//prefix++
    Integer operator++(int); //postfix++
    Integer& operator--(); //prefix--
    Integer operator--(int); // postfix--
}

Integer& Integer::operator++() {
 this->i += 1; // increment
 return *this; // fetch
}
// int argument not used so leave unnamed so
// won't get compiler warnings
Integer Integer::operator++( int ){
 Integer old( *this ); // fetch
 ++(*this); // increment
 return old; // return
```

```
}

Integer x(5);
++x; //calls x.operator++()
x++; // calss x.operator++(0)
```

- Operator `[]`
  - Must be a member function
  - Implies that the object acts like an array, so it should return a reference
  - Check for assignment to self
  - Return a refererence to `*this` .
- Assignment operator skeleton

```
T& T::operator=( const T& rhs ) {
 // check for self assignment
 if ( this =/= &rhs ) {
 // perform assignment
 }
 return *this;
}
// This checks address, not value (*this =/= rhs)
```

- Operator `()`
  - A functor, which overloads the function call operator, is an object that acts like a function.

```
struct F {
    void operator()(int x) const {
        std::cout << x << "\n";
    };
}

F f; //  f is a functor
f(2);//calls f.operator()
```

- User-defined type conversions
  - Single argument constructors
  - Conversion operations
    - Function will be called automatically
    - Return type is same as function name
  - 避免隐式转换：把复制构造函数声明为explicit,表示不能进行隐式转化。

Built-in conversions: char→short→int→float→double, T[]→T*

重载的方式

- as member function
  - 定义时少一个参数
  - 调用时只对第二个参数做类型转换
  - `=,(),[],->` 只能在类里面定义
- as global function
  - 定义时参数个数不会减少
  - 调用时两个参数都会类型转换
  - 需要在类里面声明 friend

```
//算术运算类
const T operator+ (const T &other) const;
const T operator+ (const T &l, const T &r);
//比较运算类
bool operator == (const T &other) const;
bool operator == (const T &l, const T &r);
//自增自减
const T& operator ++() { //++a
    *this += 1;
    return *this;
}
const T operator ++(int) { //a++
    T old(*this);
    ++(*this);
    return old;
}
```

Streams

A stream is an abstraction for input/output.

Think of it as a source or destination of characters of indefinte length.

Stream: Common logical interface to a device

- Stream operations
  - Extractors
    - Read a value from the stream
    - Overload the `operator >>`
  - Inserters
    - Insert a value into a stream
    - Overload the `operator<<`
  - Manipulators
    - Change the stream state
- Predefined streams
  - `cin` : standard input

    - `cout` : standard output
    - `cerr` : unbuffered error output
    - `clog` : buffered error output
- Creating a stream

```
ostream& operator<<(ostream& os, const T& obj) {
    return os;
}

istream& operator>>(istream& is, const T& obj) {
    return is;
}
```

- Other input operators
    - `int get()`
        - Returns the next character in the stream
        - Returns EOF if no characters left

        ```
        int ch;
        while((ch = cin.get()) =≠ EOF) {
            cout.put(ch);
        }
        ```

    - `get(char *buf, int limit, char delim='\n')`
- Stream Internals
    - Buffering
        - Writing to a console/file is a slow operation
        - Accumulate characters in a temporary buffer/array.
        - When buffer is full, write out all contents of the buffer to the output device at once. (known as flushing the stream)
- Stream Manipulator `#include <iomanip>`
    - Common
        - `endl` : inserts a newline and flushes the stream
        - `ws` : skip all whitespace until it finds another char
        - `boolalpha` : print true and false for bools
    - Numeric:
        - `hex` : prints numbers in hex
        - `setprecision` : adjust the precision numbers print with
    - Padding:
        - `setw` : pads output
        - `setfill` : fills padding with character

```cpp
int main() {
    cout << setprecision(2) << 1230.243 << endl; //1.2e+03
    cout << setw(20) << "OK!" ;
}
```

Templates

Function overloading

Same function name with different argument lists.

When the method is called, the compiler infers which version you mean based on the type of your parameters.

```cpp
void print(char * str, int width); // #1
void print(double d, int width); // #2
void print(long l, int width); // #3
void print(int i, int width); // #4
void print(char *str); // #5
print("Pancakes", 15); // #1
print("Syrup"); // #5
print(1999.0, 10); // #2
print(1999, 12); // #4
print(1999L, 15); // #3
```

Templates

- 复用代码的手段，generic programming（泛型编程）
- Similar operations on different types of data.

```cpp
template <typename T>
T min(T a, T b) {
    return (a < b) ? a : b;
}

int a = 3, b = 9;
int c = min<int> (a, b); // Template instantiation (T = int)
int c = min (a,b)//omit the angle brackets
```

函数模板需要实例化之和再被使用，如果没有被调用就不会被实例化

- 实例化：模板函数中的模板被替换为对应的变量类型，然后生成一个对应的函数。
- Overloading rules
  - check first for unique function match
  - Then check for unique function template match
  - Then implicit conversions on regular functions

- Class Templates
  - Classes parameterized by types

```
template <class T>
class Vector {
public:
 Vector(int);
 ~Vector();
 Vector(const Vector&);
 Vector& operator=(const Vector&);
 T& operator[](int);
private:
 T* m_elements;
 int m_size;
}

template <class T>
Vector<T>::Vector(int size): m_size(size) {
 m_elements = new T[m_size];
}
template <class T>
T& Vector<T>::operator[](int index) {
 if(index < m_size && index ⩾ 0) {
 return m_elements[index];
 } else {
 /*...*/
 }
}
```

- templates can use multiple types

```
template < class Key, class Value >
class HashTable {
 const Value& lookup (const Key&) const;
 void insert (const Key&, const Value&);
 /* ... */
}
```

- Non-type parameters

```
template <class T, int bounds = 100> class FixedVector {
public:
    FixedVector();
    // ...
    T& operator[](int);
private:
  T elements[bounds]; // fixed size array!
};
```

- Usage

```
FixedVector<int, 50> v1;
FixedVector<int, 10*5> v2;
FixedVector<int> v3; // uses default
```

显示特化、偏特化

- 隐式实例化：编译器会根据参数和模板生成一个实际的类，这个类称为模板的一个特化。
- 显示特化：模板参数是某些特殊类型时，做一些特殊的处理，编译器先找特殊情况，再找一般模板
- 偏特化：部分模板参数

Inheritance II

```
class Shape {
private:
    void prepare() {}
    void finalize() {}
public:
    Point center; //共有的成员变量

    void draw() { // 共有的成员函数
        prepare();
        doDraw();
        finalize();
    }
    virtual void doDraw() = 0; // 要求所有派生类都实现
};

class Circle : public Shape { // Circle继承自Shape, 一个 circle 是一个 shape
public:
    int radius;
    void doDraw() override {
        // draw circle
    }
};
```

基类子对象的概念

```
class Shape {
private:
    void prepare() {}
    void finalize() {}
    Point center;
public:
    Shape(int x, int y): center(x, y) {} // 构造函数
    void draw() {    // 共有的成员函数
        prepare();
        doDraw();
        finalize();
```

```
    }
    virtual void doDraw() = 0; // 要求所有派生类都实现
};

class Circle : public Shape { // Circle继承自Shape
    int radius;
public:
    Circle(int x, int y, int r): Shape(x, y), radius(r) {}
};
```

访问控制

```
class Base {
    int a;
};

struct Derived : public Base {
    void foo() {
        a++;// error: 'a' is a private member of Base
    }
};
```

类外，包括子类不能访问 Private 成员。但派生类理应看到更多的东西。受控的是访问权不是可见性。

- protected

```
class Base {
    protected:
    int a;
};

struct Derived : public Base {
    void foo() {
        a++;// error: 'a' is a private member of Base
    }
};
```

类的初始化顺序：首先将 Virtual base classes 按深度优先的顺序构造，同深度则按照 base-specifier list 从左到右的顺序构造。然后，所有的 direct base 按照 base-specifier list 从左到右的顺序初始化。然后将所有的 non-static data member 按照其在类定义中声明的顺序初始化。最后运行构造函数的函数体。

虚函数

纯虚函数不必有实现，但是也可以有。

对于一个类，如果它存在至少一个 final overrider 为纯虚的函数，则该类为抽象类。（某个接口/标准/规范）

抽象类不能用于声明对象或数据成员，但是可以作为 base class subobject; 因此也不能作为参数、返回值和转换的目标类型。

Iterators

> Provide a way to visit the elements in order, without knowing the details of the container.(Generalization of pointers)

Exception

- throw 用于抛出异常（抛出一个对象），然后离开这个大括号，直到被匹配到的 catch 捕捉到
- throw 之后的语句都不会执行，在栈中的本地变量都会被正确析构，但 throw 出来的对象直到 catch 之后才会被析构。
- try 用于执行可能抛出异常的代码，catch 用于捕获异常

```
try{
    fun1();//可能抛出异常
    fun2();//可能抛出异常
}catch(int e){//捕获int类型的异常
    cout<<"catch int"<<endl;
}catch(...){//捕获任意类型的异常
    cout<<"catch all type"<<endl;
}
```

- try 内部的语句按照从上到下的顺序执行，如果 `fun1()` 抛出异常，那么 `func2()` 不会被执行，直接进入 `catch` 块
- 被抛出的异常会逐个跟 catch 块比较，只匹配第一个符合条件的 catch 块
- catch 的时候不会进行隐式类型转换
- catch 的时候会进行派生类到基类的转换（std::exception类有很多子类，如 std::runtime_error, std::logic_error,这些都可以被catch(std::exception e)捕获）
- catch之后，异常就被捕获了，不会再继续传递给更上层的try-catch块。但是可以通过在catch块中写 throw 重新抛出异常
  - throw; 代表重新抛出当前异常
  - throw e; 代表抛出一个新的异常对象

Smart Pointer

Motivation

指针包含了太多信息

- 单个对象/数组
  - 单个对象：使用 `new, delete` ；不应该使用 `++p,--p,p[n]`
  - 数组：使用 `new[], delete[]` ；可以使用 `++p,--p, p[n]`
- 所有权

    - 所有者必须再使用完成后释放内存
    - 非所有者不允许释放内存
- 指针是否可以为空

但是指针都长一个样：`T*`。我们无法知道上述的信息。

“智能”：

- 单个对象/数组：它知道使用 delete 还是 delete[]
- 所有权：在使用完成后自动销毁，不需要手动释放
- 指针是否可以为空：某些时候可以强制要求指针不为空

Smart Pointer

- 自动回收资源
- 一些限制，例如不允许空指针
- 更多的安全限制或检查
- 能像指针一样，指向对象、执行间接访问

`std::unique_ptr`

- `#include <memory>`
- 是对象的所有者，并假设是唯一的所有者
- 能自动回收资源
- 只能移动，不能拷贝
- 为数组作了偏特化 （析构函数中默认使用 `delete[]`，并且提供了 `operator[]`）

```
std::unique_ptr<HelperType> owner ( new HelperType(inputs));
//智能指针，作用域结束后会自动 delete[],不用显示写
auto owner = std::make_unique<HelperType>(inputs);
//no raw new,更好看
```

- 所有权转移：使用移动构造/移动赋值完成

```
auto a = std::make_unique<T>;
std::unique_ptr<T> b(std::move(a));
a = std::move(b);
```

需要给一个函数传递所有权时，按值传递 `unique_ptr`.

```
std::unique_ptr<float[]> science(
    std::unique_ptr<float[]> x,
    std::unique_ptr<float[]> y, int N) {
        auto z = std::make_unique<float[]>(N);
        return z;
    }

auto result = science(std::move(upx), std::move(upy), N);
```

`std::shared_ptr`

- `#include<memory`
- 是对象的所有者，但可能有若干各 `shared_ptr` 共同指向一个对象
- 可以被拷贝
- 当指向一个对象的最后一个 `shared_ptr` 不再指向它时，对象被析构

`shared_ptr` 使用引用计数，

每一个 `shared_ptr` 的拷贝都指向相同的内存。每使用他一次，内部的引用计数加1，每析构一次，内部的引用计数减1，减为0时，自动删除所指向的堆内存。 `shared_ptr` 内部的引用计数是线程安全的，但是对象的读取需要加锁。