

无标题文档

[Smart Pointers](#)

[Class Design](#)

[Coupling | 耦合](#)

[Cohesion | 内聚 \(内部越紧密越好\)](#)

[Responsibility-driven design](#)

[Refactoring | 重构](#)

[Streams](#)

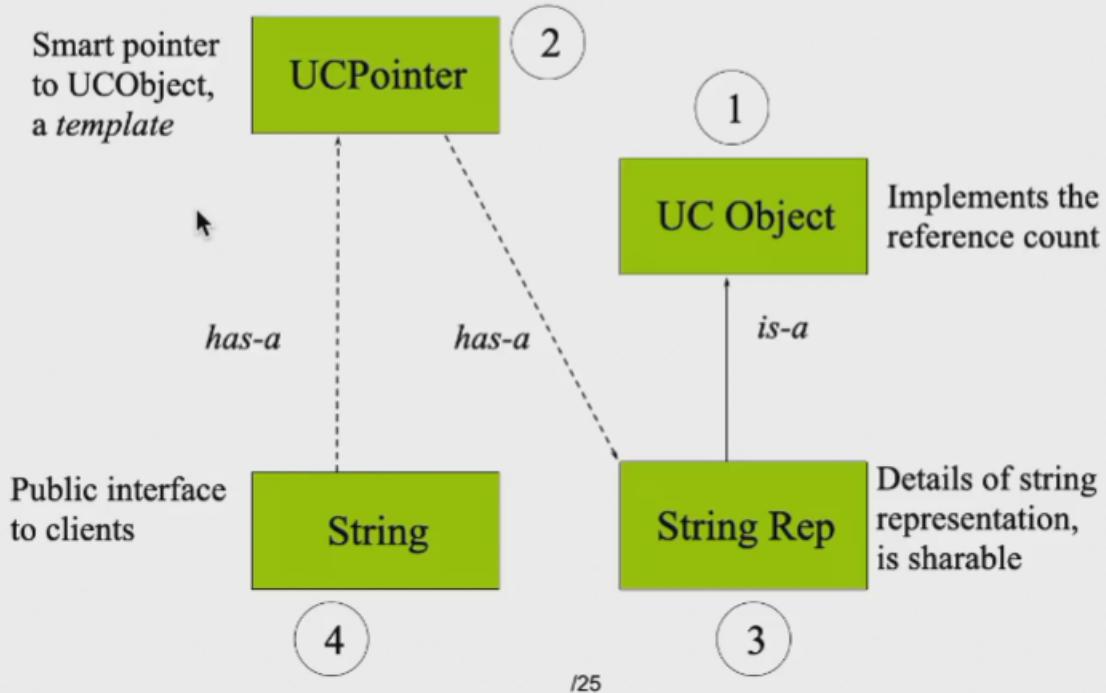
[继承构造函数](#)

[复习](#)

6.08

Smart Pointers

The four classes involved

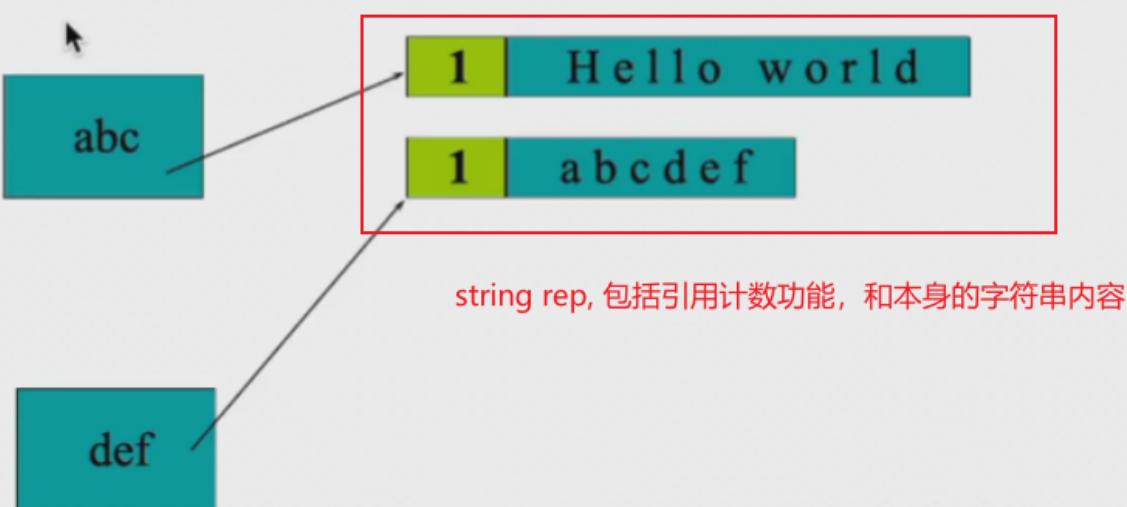


/25

UC Object 最基础的实现了 **Reference Count** 的类, 可以做 **increase** 和 **decrease** (减一减到 0 把自己消灭)

在 **UC Object** 基础上实现了 **String Rep** (继承自 **UC Object**)

UCPointer 是 **String** 里的一个指针, 指针指向 **String Rep**



/25

Reusing reference counting

```
#include <assert.h>
class UCOObject {
public:
    UCOObject() : mRefCount(0) { }
    virtual ~UCOObject() { assert(mRefCount == 0); }
    UCOObject(const UCOObject&) : mRefCount(0) { }
    void incr() { mRefCount++; }
    void decr();
    int references() { return mRefCount; }
private:
    int mRefCount;
};
```

`assert(mRefCount == 0);` (`assert` 是一个宏，当里面的东西不满足，程序就崩溃抛异常)
拷贝构造出来的 `mRefCount = 0` 而不是拷贝别人的（因制造了一个新对象，还没有被指向）

```
1 void decr()
2 {
3     mRefCount -= 1;
4     if(!mRefCount)
5     {
6         delete this;
7     }
8 }
```

- 因为析构是 `virtual`, `delete this` 在这里是动态绑定后，子类的析构会被调用

UCObject continued

```
inline void UCObject::decr() {  
    mRefCount -= 1;  
    if (mRefCount == 0) {  
        delete this;  
    }  
}
```

- "Delete this" is legal
 - But don't use *this* afterwards!

`UCPointer` 是一个模板

Class UCPointer

```
template <class T>  
class UCPointer {  
private:  
    T* m_pObj;  
    void increment() { if (m_pObj) m_pObj->incr(); }  
    void decrement() { if (m_pObj) m_pObj->decr(); }  
public:  
    UCPointer(T* r = 0) : m_pObj(r) { increment(); }  
    ~UCPointer() { decrement(); }  
    UCPointer(const UCPointer<T> & p);  
    UCPointer& operator=(const UCPointer<T> &);  
    T* operator->() const;  
    T& operator*() const { return *m_pObj; }  
};
```

| 拷贝构造只需要做 +1 的动作，因为当前并未指向任何东西所以不需要 -1

```
UCPointer(const UCPointer<T> &p)
{
    m_pObj = p.m_pObj;
    increment();
}
```

```
UCPointer &operator=(const UCPointer<T> &p)
{
    if (m_pObj != p.m_pObj)
    {
        decrement();
        m_pObj = p.m_pObj;
        increment();
    }
    return *this;
}
```

The -> Operator

- operator->() is a unary operator
 - Result must support the -> operation
- C++ allows you to overload
 - [] -- subscripting
 - () -- "function call"
 - >() -- pointer chasing
 - *() -- unary pointer dereference

The UCPointer -> operator

```
template<class T>
T* UCPointer<T>::operator->() const {
    return m_pObj;
}
```

- Example: Shape inherits from UCObject.

```
Ellipse elly(200F, 300F);
UCPointer<Shape> p(&elly);
p->render(); // calls Ellipse::render() on
elly!
```

p-> 是个语法糖,

对 p->render() p-> 是 m_pObj ,省略一个 ->

p->render() == m_pObj->render()



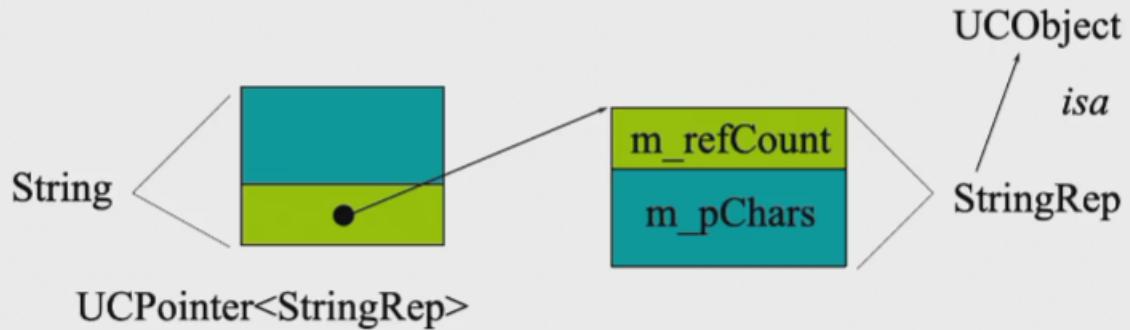
C++

复制代码

```
1 template<class T>
2 T* UCPointer<T>::operator->() const
3 {
4     return m_pObj;
5 }
6 template<class T>
7 T* UCPointer<T>::operator*() const
8 {
9     return *m_pObj;
10 }
```

Envelope and Letter

- Envelope provides protection
- Letter contains the contents



```
1 #include "UCObject.hpp"
2 #include <cstring>
3 class StringRep:public UCObject
4 {
5 public:
6     StringRep(const char *s)
7     {
8         if(s)
9         {
10             int len = strlen(s) + 1;
11             m_pChars = new char(len);
12             strcpy(m_pChars, s);
13         }
14     else
15     {
16         m_pChars = new char[1];
17         *m_pChars = '\0';
18     }
19 }
20 ~StringRep()
21 {
22     delete[] m_pChars; // 引用计数是父类要做的事情
23 }
24 StringRep(const StringRep &sr)
25 {
26     int len sr.length();
27     m_pChars = new char[len + 1];
28     strcpy(m_pChars, sr.m_pChars);
29 }
30 int length() const
31 {
32     return strlen(m_pChars);
33 }
34 int equal(const StringRep &sp) const
35 {
36     return (strcmp(m_pChars, sp.m_pChars) == 0);
37 }
38 private:
39     char *m_pChars;
40     // reference semantics -- no assignment op!
41     void operator=(const StringRep &){} // 不允许直接赋值, 因也没有赋值的必要
42 };
```

```

#include "UCPointer.h"
#include "StringRep.h"

class String
{
public:
    String(const char *s) : m_rep(0)
    {
        m_rep = new StringRep(s);
    }
    ~String() {}
    String(const String &s) : m_rep(s.m_rep) {}
    String &operator=(const String &s)
    {
        m_rep = s.m_rep; // let smart pointer do work!
        return *this;
    }
}

```

`m_rep = new StringRep(s)` `m_rep` 的类型是 `UCPointer`, `new` 的类型是 `StringRep`
 * 不能直接等, 所以会调用 `UCPointer` 里面的一个构造函数, 把 `StringRep *` 变成 `UCP`

```

UCPointer<T *r = 0> : m_pObj(r) {
    increment();
}

```

发现在 `m_rep = new StringRep(s)` 赋值前 `reference count` 就 +1, 整个赋值完成后 `reference count = 2`

但其实在 `m_rep = new StringRep(s)` 因为类型不同需要用临时对象构造 `UCP` 再赋值给 `m_rep`



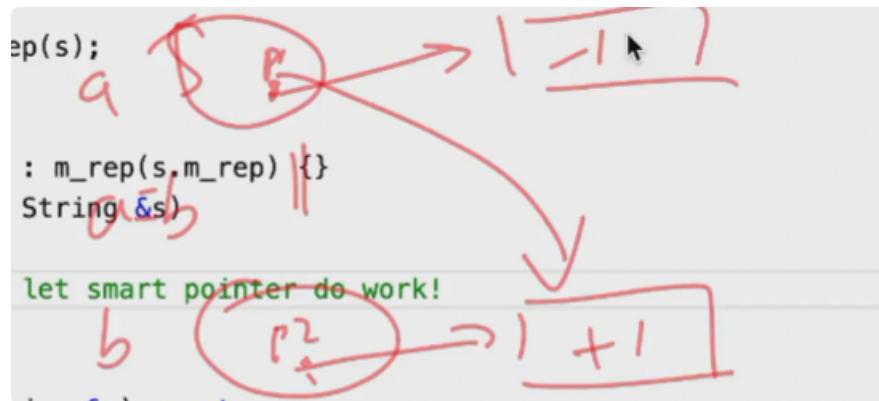
所以在赋值完成后会把临时对象析构, `reference count -= 1` -> `reference count == 1`

```

String &operator=(const String &s)
{
    m_rep = s.m_rep; // let smart pointer do work!
    return *this;
}

```

是操作两个 UCP, 此时直接发生双方引用计数的变化



```
int operator==(const String &s) const
{
    // overloaded -> forwards to StringRep
    return m_rep->equal(*s.m_rep); // smart ptr *
}
String operator+(const String &) const;
int length() const
{
    return m_rep->length();
}
operator const char *() const;
```

```
private:
    UCPointer<StringRep> m_rep;
};
```

`m_rep` 虽然表现上是一个对象，语义上是一个指针

Critique

- UCPointer maintains reference counts
- UCObject hides the details of the count String is very clean
- StringRep deals only with string storage and manipulation
- UCObject and UCPointer are reusable
- Objects with cycles of UCPointer will never be deleted

不考

Other smart pointers

- Standard library holder for raw pointers on stack
- Releases resource when destroyed (latest)

```
template <class X> std::auto_ptr {  
public:  
    explicit auto_ptr(X* = 0) throw();  
    auto_ptr(auto_ptr&) throw();  
    auto_ptr& operator=(auto_ptr&) throw();  
    ~auto_ptr();  
    X& operator*() const throw();  
    X* operator->() const throw();  
    ...  
};
```

6.15

Class Design

Designing classes

- How to write classes in a way that they are easily understandable, maintainable and reusable

Software changes

- Software is not like a novel that is written once and then remains unchanged.
- Software is extended, corrected, maintained, ported, adapted...
- The work is done by different people over time (often decades).

Change or die

- There are only two options for software:
 - Either it is continuously maintained
 - or it dies.
- Software that cannot be maintained will be thrown away.

Code quality

- Two important concepts for quality of code:
 - Coupling
 - Cohesion

Coupling | 耦合

Coupling

- Coupling refers to links between separate units of a program.
- If two classes depend closely on many details of each other, we say they are *tightly coupled*.
- We aim for *loose coupling*.

Loose coupling

- Loose coupling makes it possible to:
 - understand one class without reading others;
 - change one class without affecting others.
 - Thus: improves maintainability.

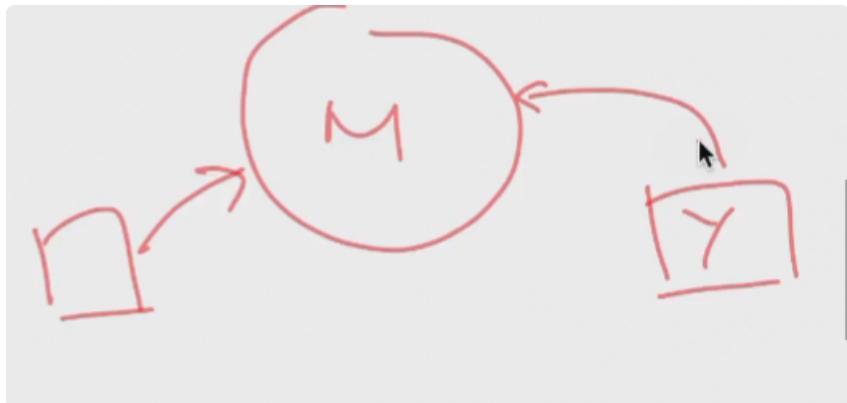
tech. to loose 两种技术

1. call-back

a. 发消息定义一个接口，另一方可以通过这个接口注册一个回调函数，当发消息时调用其回调函数从而通知自己这件事情发生了

2. message mech

a. 实现中央消息机制，把事情发给中央，中央再发布给注册者



b.

Cohesion | 内聚 (内部越紧密越好)

Cohesion

- Cohesion refers to the number and diversity of tasks that a single unit is responsible for.
- If each unit is responsible for one single logical task, we say it has *high cohesion*.
- Cohesion applies to classes and methods.
- We aim for high cohesion.

让一个单元只复杂一件事情，小到变量，大到模块…

High cohesion

- High cohesion makes it easier to:
 - understand what a class or method does;
 - use descriptive names;
 - reuse classes or methods.

| 代码复制

Code duplication

- Code duplication
 - is an indicator of bad design,
 - makes maintenance harder,
 - can lead to introduction of errors during maintenance.

Responsibility–driven design

Responsibility-driven design

- Question: where should we add a new method (which class)?
- Each class should be responsible for manipulating its own data.
- The class that owns the data should be responsible for processing it.
- RDD leads to low coupling.

把新的责任划分给谁?

把变化限制在比较小的地方

Localizing change

- One aim of reducing coupling and responsibility-driven design is to localize change.
- When a change is needed, as few classes as possible should be affected.

可扩展性 -- extensibility

- 代码不需要修改 (or 很少的修改) 就可以满足未来的需求

Refactoring | 重构

Code refactoring 是什么

在不改变软件的外部行为的条件下，通过修改代码改变软件内部结构，将效率低下和过于复杂的代码转换为更高效、更简单和更简单的代码。

Code refactoring的优缺点

优点

- 提高代码质量。
- 优化软件产品架构与性能。
- 减少项目的技术债，避免项目重写。

重构 != 重写

缺点

- 增加工作负担。
- 可能会出现一些业务上漏洞。
- 可能代码过于精炼导致代码可读性变差。

6.22

Streams

Why streams?

- Original C I/O used printf, scanf
- Streams invented for C++
 - C I/O libraries still work
- Advantages of streams
 - Better type safety
 - Extensible
 - More object-oriented
- Disadvantages
 - More verbose
 - Often slower

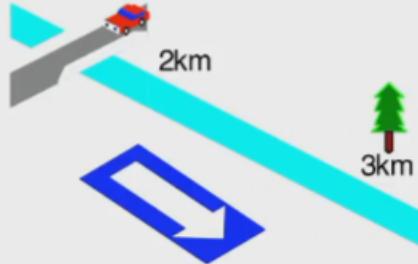
`cin cout` 速度会比 `printf scanf` 慢一点

C vs. C++

- C stdio operations work
 - Don't provide "object-oriented" features
 - No overloadable operators
- C++
 - Can overload inserters and extractors
- Moral
 - When converting C to C++, leave the I/O intact

What is a stream?

- Common logical interface to a device
- Sequential
 - There is a "position" associated with each stream
- Can
 - Produce values
 - Consume values
 - Both



Stream naming conventions

	Input	Output	Header
Generic	istream	ostream	<iostream>
File	ifstream	ofstream	<fstream>
C string (legacy)	istrstream	ostrstream	<strstream>
C string	istringstream	ostringstream	<sstream>

`cin` 是 `istream`

`istringstream` 是在字符串是建立一个流

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

Kinds of streams

- Text streams
 - Deal in ASCII text
 - Perform some character translation
 - e.g.: newline -> actual OS file representation
 - Include
 - Files
 - Character buffers
- Binary streams
 - Binary data
 - No translations

文本流和二进制流 (读文本文件里面自动把 0D0A 组合变成一个 0D(Linux))

Predefined streams

- **cin**
 - standard input
- **cout**
 - standard output
- **cerr**
 - unbuffered error (debugging) output
- **clog**
 - buffered error (debugging) output

```
cout << "hello" << endl;  
cerr << "Bye" << endl;
```

这两个是不同的流

Defining a stream extractor

- Has to be a 2-argument free function
 - First argument is an `istream&`
 - Second argument is a *reference* to a value

```
istream&  
operator>>(istream& is, T& obj) {  
    // specific code to read obj  
    return is;  
}
```

- Return an `istream&` for chaining

```
cin >> a >> b >> c;  
((cin >> a) >> b) >> c;
```

Other input operators

- `int get()`
 - Returns the next character in the stream
 - Returns EOF if no characters left
 - Example: copy input to output

```
int ch;
while ((ch = cin.get()) != EOF)
    cout.put(ch);
```
- `istream& get(char& ch)`
 - Puts the next character into argument
 - Similar to `int get();`

`cin` 中的 `get` 函数和 `getchar` 几乎一样

`istream& get` 可以放在流中使用

More input operators

- `get(char *buf, int limit, char delim = '\n')`
 - read up to limit characters, or to delim
 - Appends a null character to buf
 - Does not consume the delimiter
- `getline(char *buf, int limit, char delim = '\n')`
 - read up to limit characters, or to delim
 - Appends a null character to buf
 - Does consume the delimiter
- `ignore(int limit = 1, int delim = EOF)`
 - Skip over limit characters or to delimiter
 - Skip over delimiter if found

- `int gcount()`
 - returns number of characters just read

```
char buffer[100];
cin.getline(buffer, sizeof(buffer))
cout << "read " << cin.gcount()
     << " characters"
```
- `void putback(char)`
 - pushes a single character back into the stream
- `char peek()`
 - examines next character without reading it

```
switch (cin.peek()) ...
```

Other output operators

- `put(char)`
 - prints a single character
 - Examples

```
cout.put('a');
cerr.put('!');
```
- `flush()`
 - Force output of stream contents
 - Example

```
cout << "Enter a number";
cout.flush();
```

Formatting using manipulators

- Manipulators modify the state of the stream
 - #include <iomanip>
 - Effects hold (usually)
- Example

```
int n;  
cout << "enter number in hexadecimal"  
      << flush;  
cin >> hex >> n;
```



hex 表示从此处起，读入的东西都是十六进制的，直到其他符号恢复或程序结束

Manipulators

manipulator	effect	type
dec, hex, oct	set numeric conversion	I, O
endl	insert newline and flush	O
flush	flush stream	O
setw(int)	set field width	I, O
setfill(ch)	change fill character	I, O
setbase(int)	set number base	O
ws	skip whitespace	I
setprecision(int)	set floating point precision	O
setiosflags(long)	turn on specified flags	I, O
resetiosflags(long)	turn off specified flags	I, O

Stream flags control formatting

flag	purpose (when set)
<code>ios::skipws</code>	skip leading white space
<code>ios::left, ios::right</code>	justification
<code>ios::internal</code>	pad between sign and value
<code>ios::dec, ios::oct, ios::hex</code>	format for numbers
<code>ios::showbase</code>	show base of number
<code>ios::showpoint</code>	always show decimal point
<code>ios::uppercase</code>	put base in uppercase
<code>ios::showpos</code>	display + on positive numbers
<code>ios::scientific, ios::fixed</code>	floating point format
<code>ios::unitbuf</code>	flush on every write

set 就是将其置一，全部设置直接 `or` 即可，flag 占据不同 bit 位

Example

- A simple program

```
#include <iostream>
#include <iomanip>
main() {
    cout << setprecision(2) << 1000.243 << endl;
    cout << setw(20) << "OK!";
    return 0;
}
```

- Prints

```
1e03
```

Creating manipulators

- You can define your own manipulators!

```
// skeleton for an output stream manipulator
ostream& manip(ostream& out) {
    ...
    return out;
}
ostream& tab ( ostream& out ) {
    return out << '\t';
}
cout << "Hello" << tab << "World!" << endl;
```

Working with flags

- Code

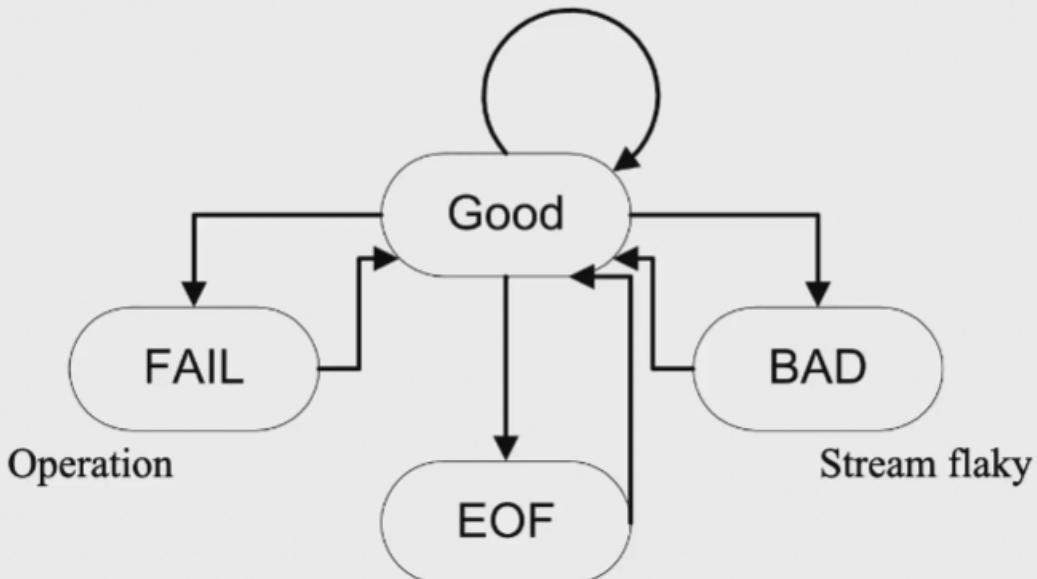
```
#include <iostream>
#include <iomanip>
main() {
    cout.setf(ios::showpos | ios::scientific);
    cout << 123 << " " << 456.78 << endl;
    cout << resetiosflags(ios::showpos) << 123;
    return 0;
}
```

- Prints

```
+123 +4.567800e+02
123
```

Stream error states

clear() returns stream to GOOD



读完会回到 EOF

Working with streams

- Error state is set after each operation
- Conversion to void* returns 0 if problem
- Can clear an error state using
 - `clear()` // Resets error state to good()
- Checking status
 - `good()` // Returns true if in valid state
 - `eof()` // Returns true if at EOF
 - `fail()` // Returns true if minor failure or bad
 - `bad()` // Returns true if in bad state

Example

```
int n;
cout << "Enter a value for n, then [Enter]" << flush;
while (cin.good()) {
    cin >> n;
    if (cin) { // input was ok
        cin.ignore(INT_MAX, '\n'); // flush newline
        break;
    }
    if (cin.fail()) {
        cin.clear(); // clear the error state
        cin.ignore(INT_MAX, '\n'); // skip garbage
        cout << "No good, try again!" << flush;
    }
}
```

`if(cin)` 中 `cin` 返回的是 `GOOD`

`FAIL` 一般是类型不匹配, `BAD` 表示流坏掉了

File streams

```
ofstream out(argv[2]);
if (!out) {
    cerr << "Unable to open file " << argv[2];
    exit(2);
}
char c;
while (in >> c) {
    out << c;
}
```

More stream operations

- `open(const char *, int flags, int)`
 - Open a specified file

```
ifstream inputS;
inputS.open("somefile", ios::in);
if (!inputS) {
    cerr << "Unable to open somefile";
    ...
}
```
- `close()`
 - Closes stream

继承构造函数

- 类具有可派生性，派生类自动获得基类的成员变量和接口（虚函数和纯虚函数）
- 基类的构造函数也没有被继承，因此：

```
class A {
public:
    A(int i) {}
};

class B : public A {           ^
public:
    B(int i): A(i), d(i) {}
private:
    int d;
};
```

- B的构造函数起到了传递参数给A的构造函数的作用：透传
- 如果A具有不只一个构造函数，B往往需要设计对应的多个透传

using 声明

- 派生类用 `using` 声明来使基类的成员函数成为自己的
 - 解决name hiding问题：非虚函数被 `using` 后成为派生类的函数
 - 解决构造函数重载问题

```
C++  
class Base {  
public:  
    void f(double) {  
        cout << "double\n";  
    }  
};  
  
class Derived : Base { //不是public继承  
public:  
    using Base::f;  
    void f(int) {  
        cout << "int\n";  
    }  
};  
  
int main()  
{  
    Derived d;  
    d.f(4);  
    d.f(4.5);  
}
```

因为没有 `public` 继承，在子类中父类的 `f` 并不存在，就算有 `public` 但由于重载，子类中还是只有一个 `f`，因为 `using` 的存在，子类中有了父类的 `f`

```

class A {
public:
    A(int i) { cout << "int\n"; }
    A(double d, int i) {}
    A(float f, char *s) {}
};

class B : A {
public:
    using A::A;
};

int main()
{
    B b(2);
}

```

- 继承构造函数是隐式声明的，如果没有用到就不产生代码

- 隐式声明的好处是不用到就不会产生实际代码，比如上述程序只会产生 A(int i) 的代码

`using A::A` 表示把所有 A 都派生掉了（using 不用给出参数表）

- 如果基类的函数具有默认参数值，`using` 的派生类无法得到默认参数值，就必须转为多个重载的函数

```

class A {
public:
    A(int a=3, double b=2.4) {}
};

```

- 实际上可以被看作是：

```

A(int, double);
A(int);
A();

```

- 那么，被 `using` 之后就会产生相应的多个函数

- 默认参数无法被派生

- 多继承且多处继承构造函数时，可能出现的重载冲突只能通过主动定义派生类的构造函数来解决
- 不能声明继承构造函数的情况：
 - 基类的构造函数是私有的
 - 派生类是从基类中虚继承的
- 一旦声明了继承构造函数，就不会再生成自动默认构造函数了

时，可能出现的重载冲突只能通过主动定义派生类的构造函数来解决

情况：

的

承的

- 就不会再生成自动默认构造函数了
 - 自己主动定义的构造优先级大于继承来的

委派构造函数

- 如果重载的多个版本的构造函数内要做相同的事情
 - 显然代码复制是设计不良的突出表现
 - 在构造函数内调用某个函数的问题是它发生在初始化之后

```
class Info {
public:
    Info() { InitRest(); } // 目标
    Info(int i) : Info() { type = i; } // 委派
    Info(char e) : Info() { name=e; }
};
```

- 目标构造函数的执行先于委派构造函数
- 构造函数不能同时委派和使用初始化列表
 - 在构造函数内赋值不是最好的选择

```
class Info {  
public:  
    Info(): Info(1, 'a') {} // 委派  
    Info(int i) : Info(i, 'a') {} // 委派  
    Info(char e) : Info(1, e) {}  
private:  
    Info(int i, char e): type(i), name(e) {} // 目标  
};
```

- 委派关系可以形成链接

```
class Info {  
public:  
    Info(): Info(1) {} // 委派  
    Info(int i) : Info(i, 'a') {} // 目标&委派  
    Info(char e) : Info(1, e) {}  
private:  
    Info(int i, char e): type(i), name(e) {} // 目标  
};
```

- 但是要防止出现循环链接
- 讲拷贝构造的时候讲过了

浅拷贝vs深拷贝

- 浅拷贝：编译器自动产生的拷贝构造函数，会执行member-wise copy
- 当有成员变量是指针时，这种拷贝是有害的

```
class C {  
public:  
    C():i(new int(0)){  
        cout << "none argument constructor called" << endl;  
    }  
    ~C(){  
        cout << "destructor called" << endl;  
        delete i;  
    }  
    int* i;  
};  
  
int main(){  
    C c1;  
    C c2 = c1;  
    cout << *c1.i << endl;  
    cout << *c2.i << endl;
```

021 Weng Kai

- 所以必须编写自己的拷贝构造函数来实现深拷贝

```
class C {  
public:  
    C():i(new int(0)){  
        cout << "none argument constructor called" << endl;  
    }  
    //增加此拷贝构造函数，根据传入的c，new一个新的int给i变量  
    C(const C& c) :i(new int(*c.i)){  
    }  
    ~C(){  
        cout << "destructor called" << endl;  
        delete i;  
    }  
    int* i;  
};
```

但

移动语义

- 拷贝函数中为指针成员分配新的内存再进行内容拷贝的方法有些时候不是必要的

3-18.cpp

```
//这是一个成员包含指针的类
class HasPtrMem {
public:
    HasPtrMem() : d(new int(0)) {
        cout << "Construct:" << ++n_cstr << endl;
    }

    HasPtrMem(const HasPtrMem& h) {
        cout << "Copy construct:" << ++n_cptr << endl;
    }

    ~HasPtrMem() {
        cout << "Destruct:" << ++n_dstr << endl;
    }

private:
    int* d;
    static int n_cstr;
    static int n_dstr;
    static int n_cptr;
};

int HasPtrMem::n_cstr = 0;
int HasPtrMem::n_dstr = 0;
int HasPtrMem::n_cptr = 0;

HasPtrMem GetTemp() {
    return HasPtrMem(); //①
}

int main(){
    HasPtrMem m = GetTemp(); //②
}

2021 WengKai 0;
```

6.29

移动语义

- 拷贝函数中为指针成员分配新的内存再进行内容拷贝的方法有些时候不是必要的

3-18.cpp

```
//这是一个成员包含指针的类
class HasPtrMem {
public:
    HasPtrMem() : d(new int(0)) {
        cout << "Construct:" << ++n_cstr << endl;
    }

    HasPtrMem(const HasPtrMem& h) {
        cout << "Copy construct:" << ++n_cptr << endl;
    }

    ~HasPtrMem() {
        cout << "Destruct:" << ++n_dstr << endl;
    }
private:
    int* d;
    static int n_cstr;
    static int n_dstr;
    static int n_cptr;
};

int HasPtrMem::n_cstr = 0;
int HasPtrMem::n_dstr = 0;
int HasPtrMem::n_cptr = 0;

HasPtrMem GetTemp() {
    return HasPtrMem(); //①
}

int main(){
    HasPtrMem m = GetTemp(); //②
}
```

©2021 Weng Kai 0;

- 构造函数被调用1次，是在①处，第一次调用拷贝构造函数是在GetTemp return的时候，将①生成的变量拷贝构造出一个临时值，来当做GetTemp的返回
- 第二次拷贝构造函数是在②处。
- 同时就有了于此对应的三次析构函数的调用
- 例子里用的是一个int类型的指针，而如果该指针指向的是非常大的堆内存数据的话，那没拷贝过程就会非常耗时，而且由于整个行为是透明且正确的，分析问题时也不易察觉。

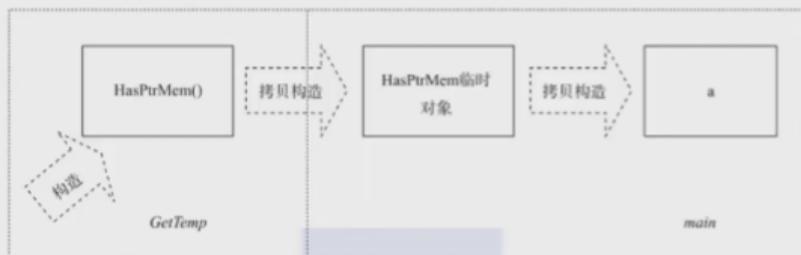


图 3-1 函数返回时的临时变量与拷贝

- 可以通过移动构造函数解决此问题

3-19.cpp

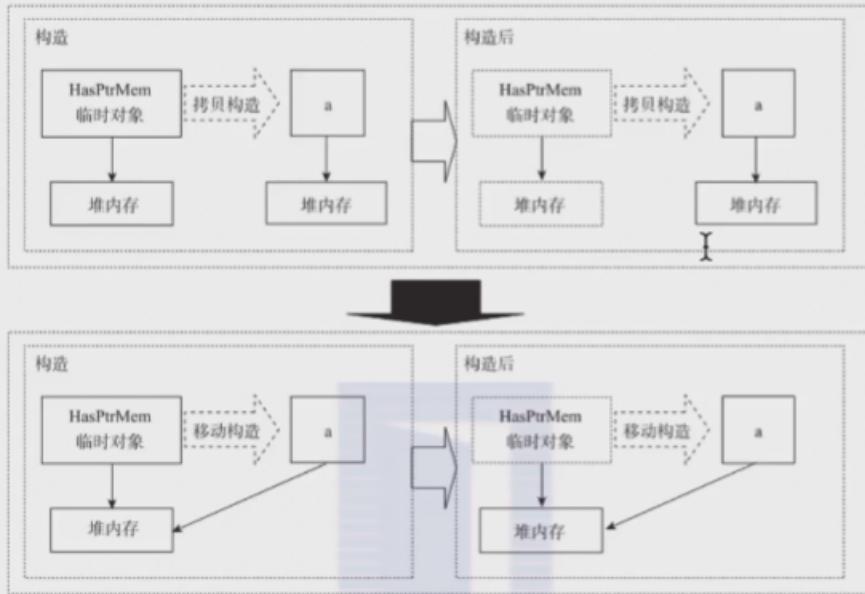


图 3-2 拷贝构造与移动构造 云栖社区 yq.aliyun.com

| 移动构造就是不复制任何的内存

(`&&` 表示右值引用)

```
HasPtrMem(const HasPtrMem& h) {
    cout << "Copy construct:" << ++n_cptr << endl;
}

HasPtrMem(HasPtrMem&& h):d(h.d) {
    h.d = nullptr; //③注意对之前的h赋空指针
    cout << "Move construct:" << ++n_mvtr << endl;
}
```

下面的是移动构造

如果不对原来的 `h` 赋空指针，可能导致 `h` 和 `d` 指向同一区域，`d` 发生了析构之后又用了 `h`

- 通过指针赋值的方式，将d的内存直接偷了过来，避免了拷贝构造函数的调用
- 注意③，这里需要对原来的d进行赋空值，因为在移动构造函数完成之后，临时对象会立即被析构，如果不改变d，那临时对象被析构时，因为偷来的d和原本的d指向同一块内存，会被释放，成为悬挂指针，会造成错误
- 为什么不用函数参数里带个指针或者引用当返回结果呢？不是性能的问题，而是代码编写效率及可读性不好

```
string *a;  
int c = 1  
int &b = c;  
Calculate(GetTemp(), b); //最后一个参数用于返回结果
```

- 移动构造函数何时会被触发
 - 一旦用到的是个临时变量，那么移动构造语义就可以得到执行
 - 在C++中如何判断产生了临时对象？如何将其用于移动构造函数？是否只有临时变量可以用于移动构造？

讲过了

左值、右值与右值引用

- 在赋值表达式中，出现在等号左边的就是“左值”，而在等号右边的，则称为“右值”
- 可以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值
- 在C++11中，右值是由两个概念构成的，一个是将亡值（xvalue, eXpiring Value），另一个则是纯右值（prvalue, Pure Rvalue）

纯右值 vs 将亡值

- 纯右值就是C++98标准中右值的概念，讲的是用于辨识临时变量和一些不跟对象关联的值。比如非引用返回的函数返回的临时变量值就是一个纯右值。一些运算表达式，比如`1 + 3`产生的临时变量值，也是纯右值。而不跟对象关联的字面量值，比如：`2`、`'c'`、`true`，也是纯右值。此外，类型转换函数的返回值、lambda表达式等，也都是右值
- 将亡值则是C++11新增的跟右值引用相关的表达式，这样表达式通常是将要被移动的对象（移为他用），比如返回右值引用`T&&`的函数返回值、`std::move`的返回值，或者转换为`T&&`的类型转换函数的返回值

右值引用

- 右值引用就是对一个右值进行引用的类型。事实上，由于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。通常情况下，我们只能是从右值表达式获得其引用。比如：

```
T && a = ReturnRvalue();
```

- 这个表达式中，假设 `ReturnRvalue` 返回一个右值，我们就声明了一个名为 `a` 的右值引用，其值等于 `ReturnRvalue` 函数返回的临时变量的值。
- 右值引用和左值引用都是属于引用类型。无论是声明一个左值引用还是右值引用，都必须立即进行初始化。而其原因可以理解为是引用类型本身自己并不拥有所绑定对象的内存，只是该对象的一个别名。左值引用是具名变量值的别名，而右值引用则是不具名（匿名）变量的别名

```
T && a = ReturnRvalue();
```

- `ReturnRvalue` 函数返回的右值在表达式语句结束后，其生命也就终结了（通常我们也称其具有表达式生命期），而通过右值引用的声明，该右值又“重获新生”，其生命期将与右值引用类型变量 `a` 的生命期一样。只要 `a` 还“活着”，该右值临时量将会一直“存活”下去
- 所以相比于以下语句的声明方式：

```
T b = ReturnRvalue();
```

- 右值引用变量声明就会少一次对象的析构及一次对象的构造。因为 `a` 是右值引用，直接绑定了 `ReturnRvalue()` 返回的临时量，而 `b` 只是由临时值构造而成的，而临时量在表达式结束后会析构因而就会多一次析构和构造的开销

- 能够声明右值引用 `a` 的前提是 `ReturnRvalue` 返回的是一个右值。通常情况下，右值引用是不能够绑定到任何的左值的。比如下面的表达式就是无法通过编译的。

```
int c;           I  
int && d = c;
```

- 相对地，在C++98标准中就已经出现的左值引用是否可以绑定到右值（由右值进行初始化）呢？比如：

```
T & e = ReturnRvalue();  
const T & f = ReturnRvalue();
```

- `e` 的初始化会导致编译时错误，而 `f` 则不会

```
T & e = ReturnRvalue();  
const T & f = ReturnRvalue();
```

- 在常量左值引用在C++98标准中开始就是个“万能”的引用类型
- 可以接受非常量左值、常量左值、右值对其进行初始化
- 而且在使用右值对其进行初始化的时候，常量左值引用还可以像右值引用一样将右值的生命期延长
- 相比于右值引用所引用的右值，常量左值所引用的右值在它的“余生”中只能是只读的
- 相对地，非常量左值只能接受非常量左值对其进行初始化

`std::move()`

- C++11中，`<utility>`中提供了函数 `std::move`，功能是将一个左值强制转化为右值引用，继而我们可以通过右值引用使用该值，用于移动语义
- 被转化的左值，其生命期并没有随着左右值的转化而改变

3-21.cpp

- 在编写移动构造函数的时候，应该总是使用 `std::move` 转换拥有形如堆内存、文件句柄的等资源的成员为右值，这样一来，如果成员支持移动构造的话，就可以实现其移动语义，即使成员没有移动构造函数，也会调用拷贝构造，因为不会引起大的问题

- 移动语义一定是要改变临时变量的值
- `Moveable c(move(a));` 这样的语句。这里的 `a` 本来是一个左值变量，通过 `std::move` 将其转换为右值。这样一来，`a.i` 就被 `c` 的移动构造函数设置为指针空值。由于 `a` 的生命期实际要到所在的函数结束才结束，那么随后对表达式 `*a.i` 进行计算的时候，就会发生严重的运行时错误

- 在C++11中，拷贝/移动改造函数有以下3个版本：

```
T Object(T&)
T Object(const T&)
T Object(T&&)
```

- 其中常量左值引用的版本是一个拷贝构造函数版本，右值引用参数的是一个移动构造函数版本
- 默认情况下，编译器会为程序员隐式地生成一个移动构造函数，但是如果声明了自定义的拷贝构造函数、拷贝赋值函数、移动构造函数、析构函数中的一个或者多个，编译器都不会再生成默认版本
- 所以在C++11中，拷贝构造函数、拷贝赋值函数、移动构造函数和移动赋值函数必须同时提供，或者同时不提供，只声明其中一种的话，类都仅能实现一种语义。

完美转发

- 完美转发(perfect forwarding)，是指在模板函数中，完全依照模板的参数类型将参数传递给模板中调用的另外一个函数

```
template <typename T>
void IamForwarding (T t) {
    IrunCodeActually(t);
}
```

- 因为使用最基本类型转发，会在传参的时候产生一次额外的临时对象拷贝
- 所以通常需要的是一个引用类型，就不会有拷贝的开销。其次需要考虑函数对类型的接受能力，因为目标函数可能需要既接受左值引用，又接受右值引用，如果转发函数只能接受其中的一部分，也不完美

- 考虑类型：

```
typedef const A T;
typedef T& TR;
TR& v = 1;
```

- 在C++11中引入了一条所谓“引用折叠”的新语言规则

TR的类型定义	声明v的类型	v的实际类型
T&	TR	A&
T&	TR&	A&
T&	TR&&	A&
T&&	TR	A&&
TR&.	TR&.	A&.

- 于是转发函数为：

```
template <typename T>
void IamForwarding (T&& t) {
    IrunCodeActually(static_cast<T&&>(t));
}
```

对于传入的右值引用：

```
void IamForwarding (X&& && t) {
    IrunCodeActually(static_cast<X&& &&>(t));
}
```

折叠后是：

```
void IamForwarding (X&& t) {
    IrunCodeActually(static_cast<X&&>(t));
}
```

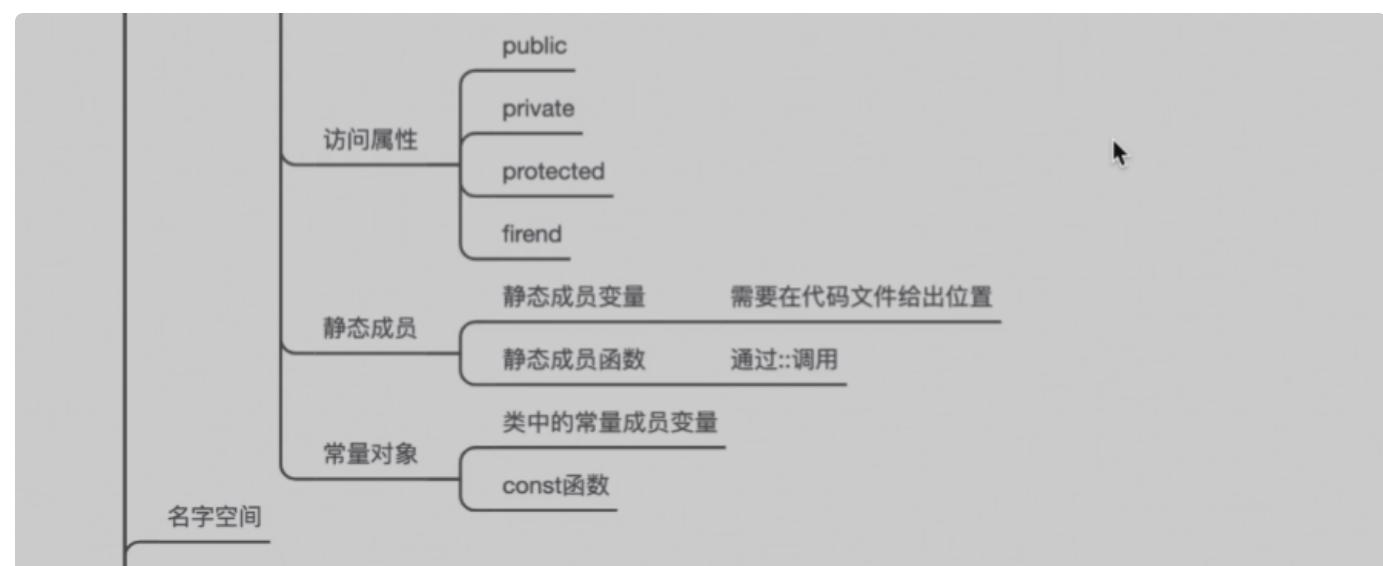
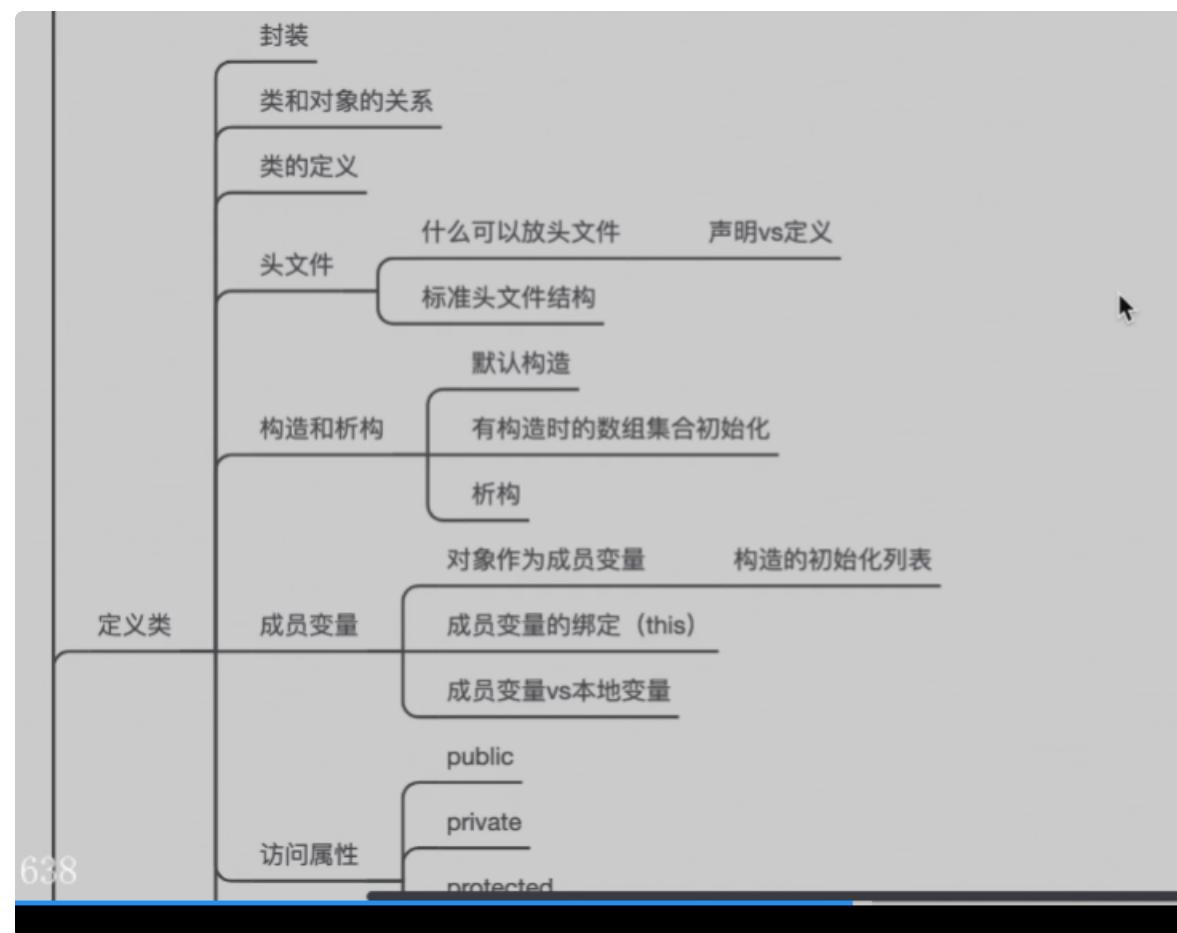
复习

程序填空：运算符重载、友元、模板、异常

堆里面的东西就是 new 和 delete

带有 [] 和没带的 new 和 delete

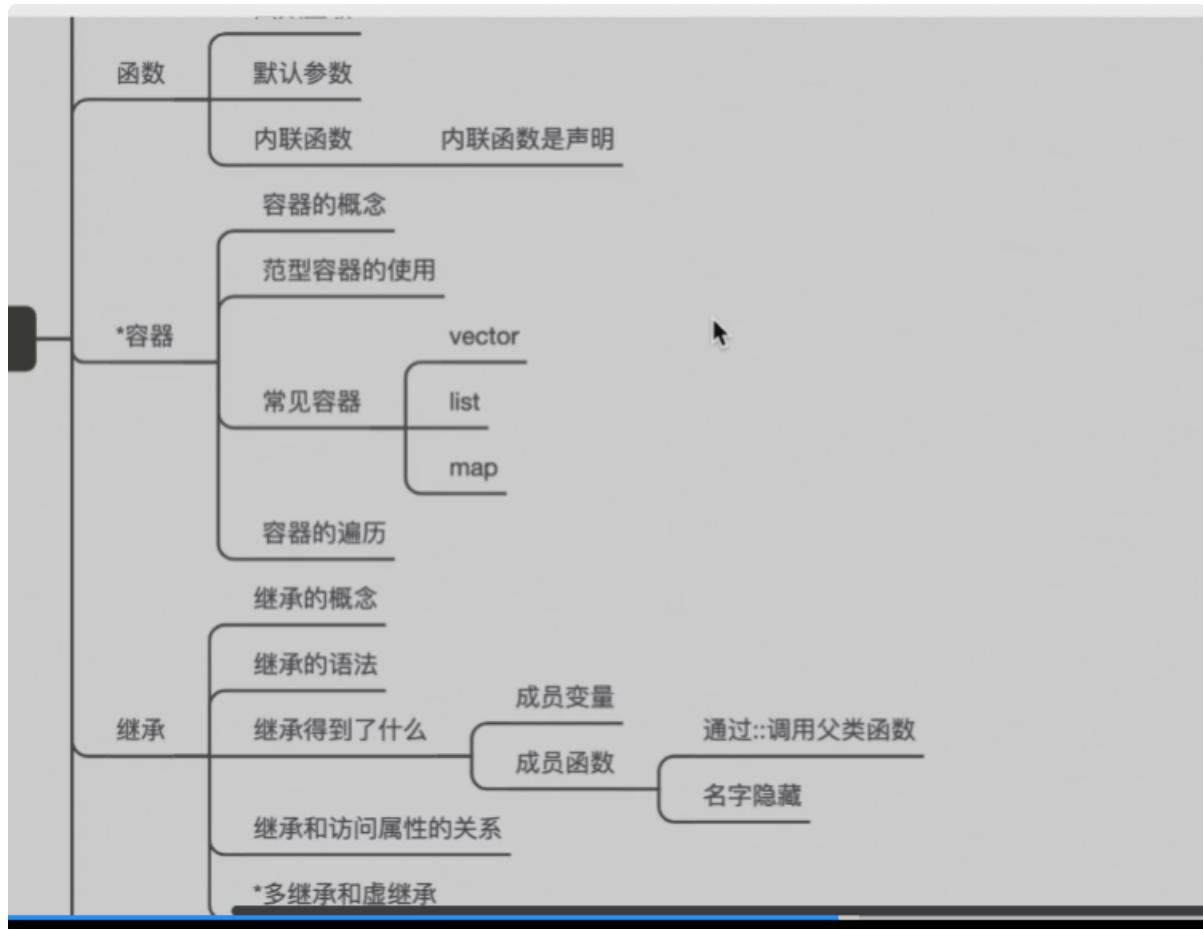
const 的位置



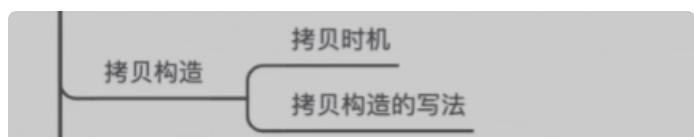
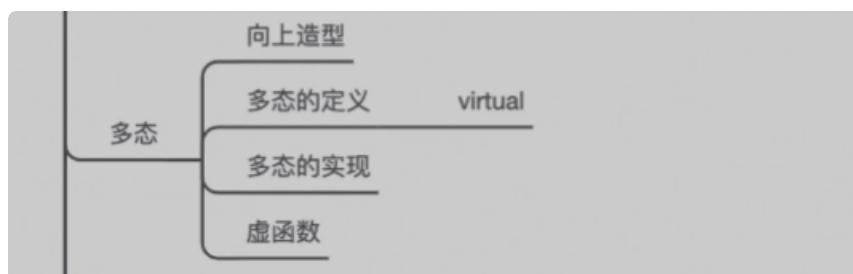
inline 和 template 必须放在头文件

函数后面加 const 表示 this 是 const

默认参数是在编译时刻起作用



vector, set, list, map



输出结果题大部分考 多态，继承、异常

父类 virtual, 儿子不写 virtual, 孙子也是 virtual 的

构造函数中调用虚函数会怎么样（在构造函数中看起来虚函数都是静态绑定）

