# 常量和常量类：

## 类中的成员变量被修饰限定是常量

Class A{

private:

const int a;

int b;

public:

void A():a(100) {……}

//当类的成员变量是一个const修饰的常量时只能在构造函数的初始化列表中进行初始化赋值

};

## 类实例被修饰限定是一个常对象

class A {

  private:

    const int a;

    int b;

  public:

    A();

    ~A();

    void setB(int n);

    void getA() const;

    void getB() const { cout << "b:" << b << endl; }

};

A::A() : a(100), b(11) {

    cout << "a:" << a << endl;

    cout << "b:" << b << endl;

}

A::~A() {}

void A::setB(int n) {

    b = n;

    return;

}

void A::getA() const {

    cout << "a:" << a << endl;

    cout << "b:" << b << endl;

}

int main(int argc, char \*argv[]) {

    A a;

    const A aa;

    // aa.setB(111);

    aa.getA();

    aa.getB();

    return 0;

}

在这个程序中，main函数中A aa；定义了一个类A的实例化对象aa; 并且使用const前缀修饰实例化的对象是**常量对象**；

这意味着实例化的对象aa不能再被修改，且使用aa进行”.”调用的成员函数，只能是那些有const后缀的成员函数。而aa.setB因为函数有赋值操作且函数没有const后缀修饰，会编译报错；

***const A aa***；

这一句就限定了实例化的aa，不能调用任何含有对类的对象有赋值修改操作的成员函数。也就是不能对其内部成员进行修改；

# 预编译,编译和链接

## 预编译g++ -E ：

#include <iostream>

using namespace std;

int main(int argc, char \*argv[]) {

    cout << "hello!" << endl;

    return 0;

}

// 如下命令预编译cpp文件为另外一个a.cpp

//  g++ -E .\预编译hello.cpp -o a.cpp

// 预编译的结果就是把源文件头上include的.h以及相关的库文件替换为对应的.h或者库文件的实际内容

使用g++ -E .\预编译hello.cpp -o a.cpp预编译后,a.的内容如下：



可以看到文件的内容增加了更多，其实预编译就是把include的.h或者库文件直接在文件中替换为其实际的内容。

从\*.c/\*.cpp –> \*.o的过程成为编译，

从\*.o->\*.exe的过程需要经过链接，所谓的链接就是把各个\*.o文件链接到一起然后生成\*.exe

# C和CPP混编

C文件的函数库无法直接在cpp中使用，需要先把.c文件使用gcc编译为.o文件然后，使用g++把.cpp和C的.o文件一起联合编译；并且在cpp文件中包含C库文件的函数头文件.h时，头文件中需要添加 extern “C” {} ，在{} 中把C库文件中的函数头包含在内部。

如下是mylib.c库文件：

int max(int a, int b) {

if (a > b)

return a;

else

return b;

}

int min(int a, int b) {

if (a > b)

return b;

else

return a;

}

以及使用mylib.c的主程序文件main.cpp

#include "mylib.h"

#include <iostream>

#include <stdio.h>

int main() {

printf("%d\n", max(23, 45));

std::cout << min(11, 20) << std::endl;

return 0;

}

编译步骤如下：  
1. gcc把.c库函数编译为.o

# gcc -c mylib.c -o mylib.o

2. 为.c库文件适配.h头文件，添加exturn “C” {}

#ifndef MYLIB\_H

#define MYLIB\_H

#ifdef \_\_cplusplus

extern "C" {

#endif

int max(int a, int b);

int min(int a, int b);

#ifdef \_\_cplusplus

}

#endif

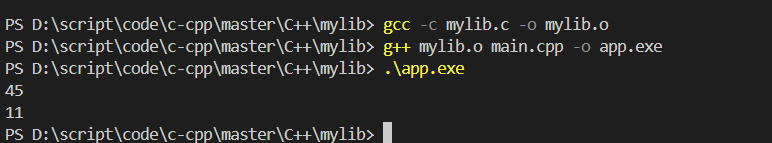
#endif

留意到，#ifdef \_\_cplusplus后只添加了 exturn “C” { 的编译语句，以及在函数头内容部分结束时，#ifdef \_\_cplusplus 只添加了结束大括号};

3. g++结合mylib.o和main.cpp联合编译

***# g++ mylib.o main.cpp -o app.exe***

得到的app.exe调用如下：



# 库文件的生成以及使用

不论是linux还是windows，库文件都是首先得到xxx.o 文件，然后在不同系统下后续的命令有所区别而已。

## CPP使用C文件或库

### 编写头文件

如果需要c文件能够在cpp中使用，那么需要为.c文件编写够被c和c++同时使用的，动态库中函数的.h头文件：

#ifndef MYLIB\_H

#define MYLIB\_H

#ifdef \_\_cplusplus

extern "C" {

#endif

int max(int a, int b);

int min(int a, int b);

#ifdef \_\_cplusplus

}

#endif

#endif

* extern "C" {}

c下编写的函数要在cpp中使用就需要用extern “C” {} 在{}中包含函数的申明语句；

* #ifdef \_\_cplusplus

用于判断是否是cpp的编译器(app的编译器会自带\_\_cplusplus宏)

如果判断是cpp的编译器，那么就给函数预编译添加上extern “C”的包含语句；

### C主文件调用动态库：

#include<stdio.h>

#include "mylib.h"

int main()

{

    printf("%d\n",max(10,20));

    return 0;

}

在这个文件中添加了#include “mylib.h”的头文件引用。

编译c程序的通用makefile如下：

CC=gcc

SRCS=calllib.c\

OBJS=$(SRCS:.c=.o)

EXE=myapp

start:$(OBJS)

    $(CC) -o $(EXE) $(OBJS) -L. -lmylib

.c.o:

    $(CC) -o $@ -c $<

clean:

    rm -fr \*.o

* SRCS=calllib.c\

主程序的源文件，如果要新增源文件，直接在SRCS后追加即可；

* EXE=myapp

编译后生成的主程序的文件名；

* $(CC) -o $(EXE) $(OBJS) -L. –lmylib

-L. –lmylib 分别是指定编译时的链接的库文件的路径，以及路径下的库文件的名称；

-L. 就是在当前路径下连接库文件，–lmylib就是连接的库文件名称为libmylib.so(linux约定编译的时候会

自动按照lib开头，so结尾的文件查找库，所以只写上去掉lib前缀和.so后缀的库文件部分名称即可）

总结：cpp调用.c的文件或者库，先得到

### Cpp主文件调用动态库：

和.c文件差异不大，就是使用的不同的编译器

#include<stdio.h>

#include "mylib.h"

int main()

{

    printf("%d\n",max(10,20));

    return 0;

}

编译cpp程序的通用makefile如下：

CC=g++

SRCS=calllib.cpp\

OBJS=$(SRCS:.cpp=.o)

EXE=myapp

start:$(OBJS)

    $(CC) -o $(EXE) $(OBJS) -L. -lmylib

.c.o:

    $(CC) -o $@ -c $<

clean:

rm -fr \*.o

只是使用的编译器CC的名称不同，其他的和c相同；

### 添加使用动态库的环境变量

Linux默认是只有so文件被放在在系统默认的lib路径下，so中的函数才能被调用，如果需要自已定义调用动态库的路径，需要添加环境变量；

所以当我们写了一个动态库以后：

* 方法1：把.so拷贝到系统的库文件看路径下,如/lib,或者/usr/lib等；
* 方法2：在~/.profile中添加如下环境变量：

export PATH=$PATH:.

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:.

在编辑追加完这两项环境变量后，还需要在命令行执行如下命令使其生效：

$ . ~/.profile

## Linux生成动态库（文件格式前后缀固定libxxx.so）

如下动态库的源文件mylib.c

int max(int a,int b)

{

        if(a>b)

                return a;

        return b;

}

1. 得到.o文件

# gcc –c mylib.c –o mylib.o –fPIC

如果是cpp，命令换成g++即可；

-fPIC是告诉编译器生成和函数定义位置无关的库。

2. 得到动态库libxxx.so（前后缀固定）

#gcc mylib.o –o libmylib.so –shared

如果是cpp命令换成g++即可

1. 和出程序文件一起，连接动态库文件，得到最终程序

# gcc main.c –o myapp.exe –I 头文件路径 –L 库文件路径 –l库文件名

如果主程序是main.cpp命令换成g++即可

如下是一个linux编译动态库.so的例子



如下是生成.so文件通用 Makefile模板：

CC=gcc

SRCS=mylib.c\

OBJS=$(SRCS:.c=.o)

MYLIB=libmylib.so

start:$(OBJS)

        $(CC) -o $(MYLIB) $(OBJS) -shared

.c.o:

        $(CC) -o $@ -c $< -fPIC

clean:

    rm -fr \*.o

其中：

* SRCS=mylinb.c\

自己编写的动态库的源文件，如果有新的源文件，直接在SRCS后追加文件名即可；

* $(CC) -o $(MYLIB) $(OBJS) –shared

这是把.o文件编译成为.so共享库文件，添加-shared后缀告诉编译器变异的是一个库文件；

* $(CC) -o $@ -c $< -fPIC

这是把-c文件编译为.o文件,**添加-fPIC即使告诉编译器生成的库文件中函数不添加偏移量**，这是生成动态库文件所必须的编译选项；

* MYLIB=libmylib.so

MYLIB是makefile编译的目标so文件；

libmylib.so是生成的.so动态库文件名称，**在liuux约定下，动态库so文件必须以lib开头，以.so结尾**；

## windows生成动态库（前后缀固定libxxx.dll）

如下mylib.c

int max(int a,int b)

{

        if(a>b)

                return a;

        return b;

}

1. 得到.o

生成.o的命令和linux类似，如下：

# gcc –c mylib.c –o mylib.o –fPIC

2. win下得到动态库libxxx.dll

# gcc mylib.o –o libmylib.dll –shared

如果是cpp文件换成g++即可

3. win下c或者cpp链接库文件得到最终程序

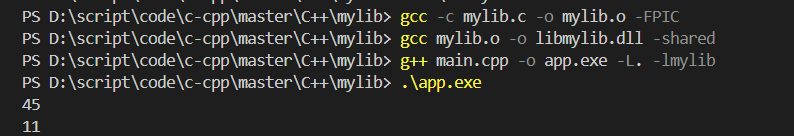
如果主程序是.c 依然使用gcc即可：

# gcc main.c –o myapp.exe –I 头文件路径 –L 要使用的库文件路径 –l要使用的库文件名

如果主程序是.cpp，命令换成g++即可：

# g++ main.c –o myapp.exe –I 头文件路径 –L 要使用的库文件路径 –l要使用的库文件名

如下是一个动态库的编译和链接使用过程：



## Windows生成静态库libxxx.lib

1. 得到.o文件

# gcc –c mylib.c –o mylib.o

如果是cpp命令换成g++即可

1. 得到win下的静态库，使用ar命令  
   #ar rcs libmylib.lib mylib.o

参数rsc分别是——r是将文件插入备存文件中，c是建立备存文件，s是索引

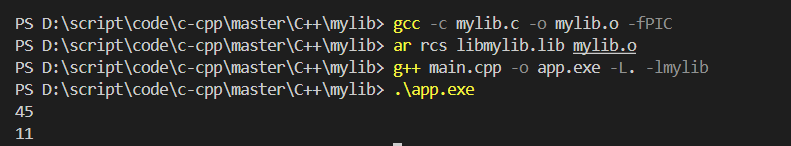
如果是多个.o文件，添加在后面即可，

1. 结合主程序使用静态库

# gcc main.c –o myapp.exe –I 头文件路径 –L 库文件路径 –l 库文件名

如果主程序是cpp,命令切换成g++即可

如下是一个win下的静态库的编译和链接使用过程：



## Linix生成静态库.a

1. 得到.o文件

# gcc –c mylib.c –o mylib.o

如果是cpp命令换成g++即可

1. 得到linux下的.a静态库，同样是使用ar命令

ar rcs libmylib.a mylib.o

参数rsc同window下的含义，分别是——r是将文件插入备存文件中，c是建立备存文件，s是索引

如果是多个.o文件，添加在后面即可，

1. 结合主程序使用静态库

# gcc main.c –o myapp.exe –I 头文件枯井 –L 库文件路径 –l 库文件名

如果主程序是cpp,命令切换成g++即可

如下是一个在linux编译和使用静态库的例子：  


## 动态库和静态库的差别

如果是连接使用的动态库，那么在编译时不会把库内容编译链接到主程序中，这样主程序的体积就会比较小。如果是静态库，在编译链接最终会把库内容加到主程序中，这样主程序体积就会比较大。

但是动态库的缺点是，因为库文件和主程序是分离的，如果主程序要正常运行，就需要把库文件放在环境中，程序能访问的地方，如果在拷贝过程中，库文件有丢失，那么换个环境，程序是无法运行的。

而静态库，因为最终的程序文件中，库文件是被编译到主程序中合而为一的，所以即便是更换了环境，只要主程序文件不损坏，拷贝过去直接就可以运行。

## 库文件使用总结：

* 不论是动态库还是静态库，第一步先要获取.o文件，这无论是在linux还是win下步骤都是一样的，在生成.o文件同时指定-FPIC参数，告诉编译器生成和定义位置无关的.o文件，如果是.c就使用gcc，如果是.cpp就使用g++;
* 对于动态库的生成，同样无论是linux还是windows，都是用编译器编译生产的，只是后缀格式不同，win下的动态库格式为dll,而linux格式是.so，且库文件名都要以lib开头，如libmylib.dll就是win下的动态库名，libmylib.so就是linux下的动态库名；
* 主程序文件要使用库文件，需要在编译时指定-L 库文件路径，-l 库文件名；这里库文件名是不带前后缀的比如win下main.cpp使用libmylib.dll,参数项就是 –lmylib即可。Linux同理
* 库文件如果是c编写的。在.o和.lib或者是.so文件都是使用gcc编译器，如果是cpp都是使用g++编译器。主程序文件如果是c编写的也同理使用gcc,如果是cpp的就使用g++编译链接。

# 类的虚函数指针和虚函数表

类的成员函数使用virtual修饰的，称为虚函数；

在类中，虚函数可以不用实现，但是只要在类中存在这没有实现的虚函数，那么该类就不能实例化，只能作为虚基类被子类继承，且在子类如果想正常实例化，就需要实现那些虚基类中没有实现的虚函数。

如下定义了一个虚基类：

#include <iostream>

using std::cout, std::endl;

class A {

public:

A();

void show();

~A();

};

A::A() { cout << "A的构造函数!\n"; }

void A::show() { cout << "A的成员函数show()\n"; }

A::~A() { cout << "A的析构函数!\n"; }

class VA {

public:

VA();

virtual void show();

virtual void info();

~VA();

};

VA::VA() { cout << "VA的构造函数\n"; }

VA::~VA() { cout << "VA的析构函数\n"; }

void VA::show() { cout << "VA的虚函数show()\n"; }

void VA::info() { cout << "VA的虚函数info()\n"; }

虚基类VA中，自己实现了所有的虚函数。

然后RA继承自VA:

class RA : public VA {

public:

RA();

~RA();

};

RA::RA() { cout << "RA的构造函数\n"; }

RA::~RA() { cout << "RA的析构函数\n"; }

RA没有重写父类VA的任何方法，所以将沿用父类的虚函数；

然后增加一个RA2继承VA,并实现部分虚函数方法：

class RA2 : public VA {

public:

RA2();

~RA2();

void show();

};

RA2::RA2() { cout << "RA2的构造函数\n"; }

RA2::~RA2() { cout << "RA2的析构函数\n"; }

void RA2::show() { cout << "RA2重写父类的show方法\n"; }

RA2只重写了父类VA的show虚函数方法，其他的将沿用父类VA的函数。

在main函数中，实例化各个类的对象：

int main(int argc, char const \*argv[]) {

A a;

A \*pa = &a;

cout << pa << endl;

RA ra;

VA \*pra = &ra;

cout << pra << endl;

RA2 ra2, ra2\_2;

cout << sizeof(a) << endl; // 1

cout << sizeof(ra)

<< endl; // 8 含有虚函数的类，会多一个虚函数表指针，用于指向虚函数表

cout << sizeof(ra2)

<< endl; // 8 虚函数表的指针大小是固定的，和虚函数的个数无关

cout << sizeof(ra2) << endl; // 虚函数表是归属于类的；

// 每个实例都有一个虚函数表指针，但同一个类的众多实例化对象，都指向同一份类中的虚函数表

cout << "\n--------main end---------\n";

return 0;

}

Main函数中，ra是RA的实例化，ra2和ra2\_2是类RA2的实例化。输出如下：

A的构造函数!

0x17d37ffb4f

VA的构造函数

RA的构造函数

0x17d37ffb40

VA的构造函数

RA2的构造函数

VA的构造函数

RA2的构造函数

1

8

8

8

--------main end---------

RA2的析构函数

VA的析构函数

RA2的析构函数

VA的析构函数

RA的析构函数

VA的析构函数

A的析构函数!

这里暂时不关心类的构造和析构，只关心输出的类实例的sizeof大小值，

从输出可以看出，对于普通的类A来说，因为没有任何的数据成员，只有一个成员函数show(),但是成员函数是不会占据空间的，所以相当于A没有实际的数据成员，其sizeof大小是1；

而RA和RA2都是从有虚函数的类VA继承来的，其RA和RA2类的实例化带有一个虚函数指针，在64位的环境下，指针大小为8。VA和RA以及RA2都没有数据成员，只有成员函数，所有实例化的对象的sizeof大小就是一个内部的虚函数表的指针大小8。

类实例中的虚函数指针只有一个，指向虚函数表，这和类具体实现了多少虚函数个数无关，RA2实现了全部的虚函数，RA只实现了一个虚函数，但是他们的实例化对象的虚函数表指针大小是一样的。

然后通过调试看下RA，RA2的类实例的虚函数表的详情：  
使用gdb可以查看实例的虚函数表的情况：

* + p ra 查看类实例ra详情
  + info vtbl ra 可以查看类实例ra的虚函数表的详情

信息如下：

(gdb) p ra

$1 = {<VA> = {\_vptr.VA = 0x7ff698805620 <vtable for RA+16>}, <No data fields>}

(gdb) info vtbl ra

vtable for 'RA' @ 0x7ff698805620 (subobject @ 0x1ae91ff5d0):

[0]: 0x7ff6988016b0 <VA::show()>

[1]: 0x7ff6988016dc <VA::info()>

(gdb) p ra2

$2 = {<VA> = {\_vptr.VA = 0x7ff698805660 <vtable for RA2+16>}, <No data fields>}

(gdb) info vtbl ra2

vtable for 'RA2' @ 0x7ff698805660 (subobject @ 0x1ae91ff5c8):

[0]: 0x7ff698801ac0 <RA2::show()>

[1]: 0x7ff6988016dc <VA::info()>

(gdb) p ra2\_2

$3 = {<VA> = {\_vptr.VA = 0x7ff698805660 <vtable for RA2+16>}, <No data fields>}

(gdb) info vtbl ra2\_2

vtable for 'RA2' @ 0x7ff698805660 (subobject @ 0x1ae91ff5c0):

[0]: 0x7ff698801ac0 <RA2::show()>

[1]: 0x7ff6988016dc <VA::info()>

从输出可以看出，类实例ra,ra2,ra2\_2的虚函数表中，虚函数都是两个，因为他们的基类VA中的虚函数个数是两个。

ra的虚函数表指针，和ra2,ta2\_2的不同，因为ra的类是RA,而ra2和ra\_2的类是RA2。虚函数表是归属于类的，某个特定的类实例化的众多对象的虚函数表指针是指向同一份虚函数表的。而不同的类，每一个类都有自己的虚函数表。

从类实例的虚函数表中的指针情况可以看出：

RA没有重新定义任何的虚函数，所以沿用的是从父类VA中继承过来的虚函数；

RA2 指重新定义了从父类继承过来的一个show()虚函数，另外一个虚函数info仍然沿用父类VA的；

总结出以下结论：

* 虚函数表附属于类，同一个类的众多实例共享一份类的虚函数表；
* 类中的虚函数表中的函数指针，指向的是具体的虚函数；虚函数表是被子类所继承的
* 只要子类没有重新定义父类的某个或者某些虚函数。那么子类的虚函数表中，会沿用从父类继承过来的这些虚函数（子类未重新定义的虚函数，虚函数表中的指针指向父类的虚函数）
* 只要子类重新定义了的虚函数，那么在子类的虚函数表中，就会使用自己新定义的函数；

# 虚函数的重载和隐藏

如下定义一个有虚函数的基类VA:

#include <iostream>

using std::cout, std::endl;

class VA {

public:

VA();

~VA();

virtual void show(int n);

virtual void show(char \*s);

virtual void show();

};

VA::VA() {}

VA::~VA() {}

void VA::show() { cout << "VA的show()方法\n"; }

void VA::show(int n) { cout << "VA的int入参show()方法\n"; }

void VA::show(char \*s) { cout << "VA的char \*入参show()方法\n"; }

然后RA类继承VA但是只实现部分的虚函数，RA2继承VA但是不实现任何的虚函数:

class RA : public VA {

public:

RA();

virtual ~RA();

void show(int n);

};

RA::RA() {}

RA::~RA() {}

void RA::show(int n) { cout << "RA的int入参show()方法\n"; }

class RA2 : public VA {

public:

RA2();

~RA2();

};

RA2::RA2(){}

RA2::~RA2(){}

Main函数如下：

int main(int argc, char const \*argv[])

{

RA ra1;

ra1.show(2);

// ra1.show(); //子类没有把所有的重载show()重新定义，所以无法主动调用父类的重载版本

RA2 ra2;

ra2.show(); //因为子类中没有实现任何重载版本的show虚函数，所以可以访问基类的重载方法

ra2.show(2);

ra2.show("abcdefg");

return 0;

}

输出：

RA的int入参show()方法

VA的show()方法

VA的int入参show()方法

VA的char \*入参show()方法

从输出可以看出，RA因为自己重新定义了int入参的虚函数show。所以通过实例ra1调用时如果调用方法中带int参数，就会调用自己实现的方法。

但是RA1没有实现其他重载的show方法，比如无参的show方法，子类中没有实现，如果调用就会出错。而这在普通的类继承中，如果子类没有实现，那么会主动调用父类的，但是在含有重载虚函数的类继承中，这种情况不成立，一旦子类对父类的重载版本的虚函数只实现了一部分，那么子类就只能调用自己已经实现了的方法，未实现的重载版本仍然在父类中，但是子类无法调用。

如果想要完全可以自动调用父类的所有重载版本的虚函数方法，那么子类就不要实现任何的父类的重载版本虚函数。如本例中的RA2没有实现任何的VA的虚函数，它的实例化对象ra2就可以调用父类的虚函数方法。

可以得出以下结论：

* 也就是只要子类一旦定义了基类中的重载版本的某个虚函数，那么只能访问子类自己重写的这些函数，同时这样也会切断子类访问父类的重载版本函数的调用；
* 如果子类没有重新定义任何的基类的重载版本的虚函数，那么子类可以调用父类的重载版本的虚函数；
* 或者是，子类把父类的所有重载版本都一一重新实现，这样也可以调用自己的所有重载版本的虚函数；

# 可以把析构函数申明为虚函数

类的析构函数可以被定义为虚函数，但是这样会占据内存，一般我们不把类的析构函数定义为虚函数。

但是如果作为虚基类的话，为了能够让基类的指针指向子类new出的对象时，在析构时也能够调用子类的析构函数，最好是把虚基类的析构函数申明为虚函数。

如下是两个例子：

#include <iostream>

using std::cout, std::endl;

class VA {

public:

VA() { cout << "VA的构造函数!\n"; }

virtual ~VA() { cout << "VA的析构函数\n"; }

};

class RA : public VA {

public:

RA() { cout << "RA的构造函数\n"; }

~RA() { cout << "RA的析构函数\n"; }

};

class A {

public:

A() { cout << "A的构造函数\n"; }

~A() { cout << "A的析构函数\n"; }

};

class B : public A {

public:

B() { cout << "B的构造函数\n"; };

~B() { cout << "B的析构函数\n"; };

};

int main(int argc, char const \*argv[]) {

VA \*p = new RA();

A \*pa = new B();

delete p; // RA继承自VA,父类VA的析构函数是虚函数，虽然使用父类的指针指向子类new出的对象，但是析构时会自动调用子类的析构函数

delete pa; //这里普通类B继承自普通类A,

// pa是使用的父类A的类型指针指向子类的new出的对象，析构时不会调用子列的析构函数，只会调用父类A的

return 0;

}

输出：

VA的构造函数!

RA的构造函数

A的构造函数

B的构造函数

RA的析构函数

VA的析构函数

A的析构函数

在这个例子中，RA继承自虚基类VA,而B继承自普通基类A。

当在实际使用时，如果用父类的指针指向子类new出的对象，普通类在delete p销毁对象时，不会调用子类的析构函数，只会调用父类的。本例中使用普通类父类A的指针pa指向了子类Bnew出的对象，在delete pa时，子类B的析构函数不会被调用。

但是使用父类VA指针p指向new出的子类RA对象时，delete p时子类的析构函数也可以被正常调用。这就是虚函数在运行时的多态的表现。

一般地, 如果基类中是含有virtual函数的虚基类，尽量把基类的析构函数也声明为虚函数，这样可以避免当父类对象指向new出的子类时，在析构中不调用子类的析构函数的情况出现。

# 类的菱形继承问题

菱形继承是值，在一开始有某个基类A，被超过1个的子类B,C继承,后续又有新的子类同时继承了B,C。体的继承关系结构像一个菱形，结构如下：  
 A

/ \

B C

\ /

D

存在的问题：因为B,C都有A的数据成员，这会导致D在访问父类时不知道访问哪一边。

如下的一个多继承的菱形继承问题：

class A {

public:

int num = 0;

A(int n) : num(n) {}

A() {

} // 如果是带参初始化参数列表的情况，在菱形继承中，需要原始的基类有无参的构造函数

~A(){};

};

class B : public A {

public:

B(int n) : A(n){};

~B(){};

};

class C : public A {

public:

C(int n) : A(n){};

~C(){};

};

// D同时继承B,C

class D : public B, public C {

public:

D(int n) : B(n), C(n){};

~D(){};

};

在主函数中，用如下的代码调用查看数据的结构：

A a(1);

B b(2);

C c(3);

D d(4);

cout << "a.num:" << a.num << endl; // a.num:1

cout << "b.num:" << b.num << endl; // b.num:2

cout << "c.num:" << c.num << endl; // c.num:3

// cout << "d.num:" << d.num << endl;

// //报错，d.num不明确没需要指明访问那个父类的变量

cout << "d.B::num:" << d.B::num << "addr d.B::num:" << &d.B::num

<< endl; // d.B::num:4addr d.B::num:0xf685fff77c

cout << "d.C::num:" << d.C::num << "addr d.C::num:" << &d.C::num

<< endl; // d.C::num:4addr d.C::num:0xf685fff780

输出：

a.num:1

b.num:2

c.num:3

d.B::num:4addr d.B::num:0x775c9ff59c

d.C::num:4addr d.C::num:0x775c9ff5a0

从输出可以看出，最终的类D，其实例化对象d1,如果是要访问集成的num数据成员，直接写d1.num会有不明确的报错。需要写明完成的访问路径才行，如d1.C::num 和 d1.B::num这种形式。

而且从两者的取地址可以看出，d1.C::num 和 d1.B::num是两个单独的区域，也就是说D同时继承B和C的时候，B中有一个num，C中也有一个num,。这种菱形继承存在着资源的浪费，因为只要是开始的基类A中有的数据成员，在D被继承下来后，都会有两份分别存在B，C中。

解决直接访问冲突和内存浪费的问题，如果是继承共同的祖先，那么声明继承为虚继承即可，就是在集成式提那家关键字virtual，如下：  
class D:virtual public A{…} 或者是 classD:public virtual A{…} virtual和public关键字顺序无关紧要。

下面是使用虚继承的例子：

// 如果多继承的时候，统计父类之间有共同的祖先，那么使用虚继承

class B1 : virtual public A {

public:

B1(int n) : A(n) {}

~B1(){};

};

class C1 : virtual public A {

public:

C1(int n) : A(n) {}

~C1(){};

};

class D1 : public C1, public B1 {

public:

D1(int n) : C1(n), B1(n){};

~D1(){};

};

如下调用分别观察虚继承的数据情况：

B1 b1(5);

C1 c1(6);

D1 d1(7);

cout << "c1.num:" << c1.num << endl; // c1.num:6

// 如下观察数据成员num的地址：

cout << "d1.num:" << d1.num << "addr d1.num:" << &d1.num << endl;

cout << "d1.C1::num:" << d1.C1::num << "d1.C1::num:" << &d1.C1::num << endl;

cout << "d1.B1::num:" << d1.B1::num << "d1.B1::num:" << &d1.B1::num << endl;

cout << "d1.C1::A::num:" << d1.C1::A::num

<< "d1.C1::A::num:" << &d1.C1::A::num << endl;

cout << "d1.B1::A::num:" << d1.B1::A::num

<< "d1.B1::A::num:" << &d1.B1::A::num << endl;

输出：

d1.num:0addr d1.num:0x62e5bffba0

d1.C1::num:0d1.C1::num:0x62e5bffba0

d1.B1::num:0d1.B1::num:0x62e5bffba0

d1.C1::A::num:0d1.C1::A::num:0x62e5bffba0

d1.B1::A::num:0d1.B1::A::num:0x62e5bffba0

可以看出，使用菱形继承后，d1.num、d1.C1::num、d1.B1::num、d1.C1::A::num、d1.B1::A::num无论使用哪种方式访问数据成员num，地址都是一样的，也就是谁成员num就只有一份。

这样就解决了菱形继承时的存储浪费的问题。

注意：因为是虚继承，参数列表传递只在D1的最近一层父类进行，对于再上层的基类，会调用不会带参的构造函数。所以需要原始基类A具有无参的构造函数，用于在构造过程中无参构建初始基类A；

# 设计模式

## 面向对象的设计原则

* 单一职责原则：类的职责单一，对外只提供一种功能，而引起类变化的原因都是只有一个；
* 开闭原则：类改动通过增加代码进行，而不是修改源代码；
* 李氏代还原则：人格抽象类出现的地方都可以用他的实现类进行替换，实际就是虚拟机制，语言级别实现面向对象功能；
* 依赖倒转原则：依赖于抽象（接口），而不要依赖于具体的实现（类），也及时针对接口编程；
* 接口隔离原则：不应强迫用户程序依赖他们不需要的额借口方法，一个接口应该肢体同一种对外功能，不应把所有操作都放在一个接口里；
* 合成复用原则：如归使用集成，会导致父类的任何变换都影响子类。如果使用对象组合，就降低了这种依赖关系，对于继承和组合，优先使用组合。
* 迪米特原则：又称为最少知识原则。一个对象应该对其他对象尽可能少的了解，从而降低各个对象之间的耦合，提高系统可维护性，例如在一个程序中，各个模块相互调用，通常会提供一个统一的接口实现，这样其他模块无需了解拎一个模块的内部细节。这样当一个模块发生变化时，不会影响其他模块的使用（黑盒原理）。

### 开闭原则

如下以一个计算器类为例子，普通的实现方式如下：

class Cacculator {

private:

  int a, b;

  string op;

public:

  Cacculator(int a, int b, string oper) {

    this->a = a;

    this->b = b;

    this->op = oper;

  }

  ~Cacculator(){};

  int getResult() {

    if (op.compare("+") == 0)

      return a + b;

    else if (op.compare("-") == 0) {

      return a - b;

    } else if (op.compare("\*") == 0) {

      return a \* b;

    } else if (op.compare("/") == 0) {

      return a / b;

    }

  }

};

在这个例子中，所有的计算方当时的实现都是在成员函数函数getResult中进行的。不便之处在于，瑞国后续想要新增或者修改功能的话，会动类成员函数的源码，这不符合开闭原则。

可以把“计算”概念抽离出来，定义一个抽象基类，用抽象基类的虚函数来限定计算所需要的接口。然后每种具体的计算，实现时继承抽象基类的接口，分别针对具体的计算方式实现自己的计算细节即可。如下：

// 抽象类定制计算器接口

class Calc {

public:

  virtual int getResult() = 0;

  virtual void setOpNums(int a, int b) = 0;

  virtual ~Calc() { cout << "Calc 析构\n"; };

};

// 各个计算类型依照抽象类，实现自己的操作

class AddCalc : public Calc {

public:

  int a, b;

  void setOpNums(int a, int b) {

    this->a = a;

    this->b = b;

  }

  virtual int getResult() { return a + b; }

  AddCalc() { cout << "AddCalc 构造\n"; };

  ~AddCalc() { cout << "AddCalc析构\n"; };

};

class SubCalc : public Calc {

public:

  int a, b;

  void setOpNums(int a, int b) {

    this->a = a;

    this->b = b;

  }

  virtual int getResult() { return a - b; }

};

class MulCalc : public Calc {

public:

  int a, b;

  void setOpNums(int a, int b) {

    this->a = a;

    this->b = b;

  }

  virtual int getResult() { return a \* b; }

};

这样把各种类型的计算方式抽象抽离出来，一方面是可以把最小粒度的操作队里出来，另一方面是方便功能的修改。从这个例子中的计算器类可以看出，当腰增加功能时，直接继承抽象基类Calc即可，然后每种计算方式的实现的代码结构都大同小异，扩展方便。

用如下函数调用检验：

int main(int argc, char const \*argv[]) {

  Calc \*cal = new AddCalc;

  cal->setOpNums(5, 6);

  cout << cal->getResult() << endl;

  unique\_ptr<Calc> ucalc(cal); // 这里使用智能指针

  ucalc = move(

      make\_unique<MulCalc>()); // 智能指针重新赋值需要move移动语义转让所有权

  ucalc->setOpNums(11, 12);

  cout << ucalc->getResult() << endl;

  return 0;

}

输出：

AddCalc 构造

11

AddCalc析构

Calc 析构

132

Calc 析构

注意到为了实现能够在使用基类指针，指向子类的new出的对象时，能够调用子类的析构函数，这里采用了把寄来Calc的析构函数也使用virtual设置为虚函数的方法。同时在数函数中使用了uuique\_ptr智能指针包装创建出来的Calc子对象实例，这样不用手动delete，在离开作用域时，会自动调用析构函数。

### 迪米特原则

又称为最小值知识原则，当客户想要某个功能或者某些信息时，面对原生的类可能信息多而且杂乱，那么通过一个中专的中间类，作为接口的功能，让这个中转的类来访问原生类，而用户调用中转类的时候，只需要提供需要的信息条目即可，不用让用户名对原生类的多杂的信息。

如下以楼盘信息为例，当用户需要去买房时，面对各种各样的楼盘类型，高精装，毛坯房的，预售等等。如果是自己一个个去了解，就比较耗时，这时候就有了中介这个职业，用户只需要告诉中介需要什么样的房子，那么中介会帮用户筛选后，返回给用户需要的信息…

class AbstractBuild {

public:

  virtual void sale() = 0;

  virtual string getQuantity() = 0;

};

class BuildingA : public AbstractBuild {

private:

  string quantity = "";

public:

  BuildingA() { quantity = "高品质楼盘"; }

  virtual void sale() { cout << "楼盘" << quantity << "被售卖" << endl; }

  virtual string getQuantity() { return quantity; }

};

class BuildNormal : public AbstractBuild {

private:

  string quantity = "";

public:

  BuildNormal() { quantity = "普通楼盘"; }

  virtual void sale() { cout << "楼盘" << quantity << "被售卖" << endl; }

  string getQuantity() { return quantity; }

};

上面是楼盘的类型，依赖于共同的抽象基类的接口AbstractBuild，其中定义了建筑是否在售，建筑类型获取的接口。

如下是中介类：

// 中介类

class Mediator {

private:

  std::vector<AbstractBuild \*> resources;

public:

  Mediator() {

    AbstractBuild \*building = new BuildingA();

    resources.push\_back(building);

    building = new BuildNormal();

    resources.push\_back(building);

  }

  ~Mediator() {

    for (std::vector<AbstractBuild \*>::iterator it = resources.begin();

         it != resources.end(); it++) {

      if (\*it != NULL) {

        delete \*it;

      }

    }

  }

  // 对外暴露的接口

  AbstractBuild \*findBuild(string quantity) {

    for (std::vector<AbstractBuild \*>::iterator it = resources.begin();

         it != resources.end(); it++) {

      if ((\*it)->getQuantity() == quantity)

        return \*it;

    }

    return NULL;

  }

};

在中介类中，掌握着所有的楼盘信息的资源，在成员resources中，然后中介类对外暴露的接口是findBuild，用于提供给客户，让客户通过调用该成员函数并传递要求，获取可用的楼盘信息。

在这个例子中，用户无需关心具体的楼盘的细节，只需要和中介类打交道即可，也就是用户获得经可能少的原生资源的信息。

### 合成复用原则

两种方式实现合成复用：类继承和类组合。一般在尽可能的情况下，使用类组合的方式。

如下是一个支付类的例子，有若干的支付渠道，支付包，微信受虚基类Payment的管控，而其他的支付BankPay和CardPay不受Payment的管控，现在需要对这些支付方式进行适配：

class Payment {

public:

  virtual void pay(int n) = 0;

};

class Alipay : public Payment {

public:

  void pay(int n) { cout << "支付宝支付" << n << "元\n"; };

};

class Wechat : public Payment {

public:

  void pay(int n) { cout << "微信支付" << n << "元\n"; }

};

class BankPay {

public:

  void cost(int n) { cout << "银行支付" << n << "元\n"; }

};

class CardPay {

public:

  void cost(int n) { cout << "刷卡支付" << n << "元\n"; }

};

如下是使用继承适配BankPay和CardPay:

class AdaptorBankPay : public Payment, public BankPay {

public:

  void pay(int n) { this->cost(n); }

};

class AdaptorCardPay : public Payment, public CardPay {

public:

  void pay(int n) { this->cost(n); }

};

适配之后，使用适配的类，然后所有的支付方式都可以听过pay这个方法实现支付，如下：

int main(int argc, char const \*argv[]) {

  Payment \*payment = new Alipay();

  payment->pay(10);

  delete payment;

  payment = new AdaptorBankPay();

  payment->pay(20);

}

输出：

支付宝支付10元

银行支付20元

上述是使用类继承的方式，但是缺点是，如果每新增有支付方式，就需要同时定义一个新的支付适配器。

但使用类组合的形式就更简洁一些,如下是新定义一个移动支付的方式MobilPay，组合AliPay和WechatPay,只需要在MobilPay中组合一个ApiPay和WechatPay的基类型Payment的指针即可：

/\* 方式2：使用类组合的方式复用 \*/

class MobilePay {

  // 移动支付类

public:

  Payment \*inner\_pay = nullptr;

  MobilePay(Payment \*payment) { inner\_pay = payment; };

  void pay(int n) { this->inner\_pay->pay(n); }

};

调用如下：

  MobilePay \*mpay = new MobilePay(payment);

  mpay->pay(100);

  Wechat \*payment2 = new Wechat();

  MobilePay \*mpay2 = new MobilePay(payment2);

mpay2->pay(90);

输出：

银行支付100元

微信支付90元

使用组合的时候的限制条件需要被组合的那些类，有共同的祖先最好，否则类型定界是个问题，如本例中，因为Alipay和WechatPay有共同的Payment祖先类，所以组合才能畅通无阻；

## 简单工厂模式

有时候类的实现细节，如需要读取文件配置，需要传递构造函数等等。对于调用者来说并不关心，调用者关心的是获取到类实例的便捷性。通过调用一个类或者方法，就能立即获得可用的对象。

简单工厂模式就是用于面向调用者时，及时的给调用者返回需要的特定类实例，其存在是向调用者隐藏了具体类的实现过程(不过用户也不关心具体的实现过程就是了).

如下列，一个水果工厂的例子，为说明问题这里简化了类的结构。

这里先定义了一个水果类的抽象基类，然后所有的水果类都继承自抽象基类。然后又定义了一个水果工厂类，其中由成员函数CreateFruit接受传入的参数类型，从而去创建具体的水果类实例。

class AbstractFruit {

public:

  virtual void showName() = 0;

};

class Apple : public AbstractFruit {

public:

  Apple(){};

  void showName() { cout << "苹果" << endl; }

};

class Banana : public AbstractFruit {

public:

  Banana(){};

  void showName() { cout << "香蕉" << endl; }

};

// 水果工厂类

class FruitFactory {

public:

  static AbstractFruit \*CreateFruit(string t) {

    if (t == "苹果")

      return new Apple();

    if (t == "香蕉")

      return new Banana();

  }

};

调用如下：

int main(int argc, char const \*argv[]) {

  FruitFactory \*factory = new FruitFactory();

  AbstractFruit \*f1 = factory->CreateFruit("苹果");

  f1->showName();

  delete f1;

  AbstractFruit \*f2 = factory->CreateFruit("香蕉");

  f2->showName();

  delete f2;

  return 0;

}

输出：

  苹果

  香蕉

简单工厂的工厂类返回的是具体的类实例，而入参是由用户设定的足够少的参数用于确定产品类型，但是简单工厂模式不太符合开闭原则，因为如果扩展产品类型的时候，会需要直接修改工厂类的源码。

但是好的一点是，工厂类的存在，在调用者层面，无需知道具体产品的实现细节，只需要调用工厂类的方法即可。

## 从简单工厂到工厂方法模式

既然简单工厂不符合开闭原则，扩展的时候需要修改工厂类或者成员函数的源码，那么如果能够实现新增产品类型的时候，直接新增工厂类而不是修改工厂类的方式来实现工厂模式不就行了？

工厂方法模式就是这个作用，因为在简单工厂模式中，只有对产品的抽象，然后是具体的产品都是继承自产品的抽象基类。那么同理，只要工厂类也有抽象基类就可以了。定义一个抽象的工厂的虚基类，然后在其中定制接口。然后具体的产品如果要扩展时，在新增产品类的同时新增对应的工厂类即可。

如下，在简单工厂的基础上，增加了工厂类的抽象，这样当有新的水果类时，直接新增对应类型的水果工厂类即可：

class AbstractFruit {

public:

  virtual void showname() = 0;

};

class Apple : public AbstractFruit {

public:

  void showname() { cout << "苹果" << endl; };

};

class Banana : public AbstractFruit {

public:

  void showname() { cout << "香蕉" << endl; };

};

class AbstractFactory { //新增的工厂抽象基类

public:

  virtual AbstractFruit \*createFruit() = 0;

};

class AppleFactory : public AbstractFactory {

public:

  AbstractFruit \*createFruit() { return new Apple(); };

};

class BananaFactory : public AbstractFactory {

public:

  AbstractFruit \*createFruit() { return new Banana(); }

};

调用如下：

int main(int argc, char const \*argv[]) {

  AppleFactory \*appleFac = new AppleFactory();

  AbstractFruit \*apple = appleFac->createFruit();

  BananaFactory \*bananaFac = new BananaFactory();

  AbstractFruit \*banana = bananaFac->createFruit();

  apple->showname();

  banana->showname();

  return 0;

}

输出：

苹果

香蕉

但是工厂方法模式带来了新的问题，那就是当扩展的具体类过多时，类的数量就会成倍的增加，不利于维护。

## 抽象工厂模式

不同于工厂方法模式，抽象工厂模式返回是按照产品族的类型，对产品族的工厂类的抽象，以及产品的抽象，用于返回一组产品的组合，也就是说抽象的维度增加了，不再是单个维度的抽象。如下仍然以水果类为例子。

比如产地在中国的工厂可以产出苹果，香蕉，梨，同样美国的工厂也可以产出苹果，香蕉，梨。这些工厂是有着类似的产品种类组合的产出的。那么可以在工厂中集合这些产品的制造方式，然后分别以成员函数的方式实现。比如中国的工厂有自己生产苹果，梨等的接口，美国的工厂也有自己的生产苹果，梨的接口，但是这些接口是类似的。

而各个工厂的不同之处在于，一个工厂生产的一批产品中组合了各种区域内的各种产品，比如美国的工厂生产的是美国的苹果，梨，香蕉，中国的工厂生产的是中国的苹果，梨，香蕉…

从中可以提炼出的抽象概念是：

每个工厂都有类似的产品生产接口，如生产苹果，生产梨，生产香蕉…,并且是把这些生产接口都合并为工厂的成员函数，然后工厂制造的具体的产品实例，如苹果，梨等等，由工厂的成员函数返回。

各个工厂生产出来一批的产品，在二次分类上有所不同，如美国的苹果，中国的苹果，美国的梨，中国的梨等等...

这需要两个维度的抽象：

* 产品种类的抽象：因为不同的工厂都能产出苹果，所以对于苹果这个产品大类，不论是美国的，中国的，还是日本的，反正产出的都是苹果，所以都可以用上层抽象。
* 工厂类型的抽象：不同的工厂，都具有类似的接口，如生产苹果的接口，生产梨的接口等，所以工厂类也是具有多种生产方式的上层的抽象。

如下先是对产品的抽象以及具体的产品类：

// 抽象的苹果

class AbstractApple {

public:

  virtual void showName() = 0;

  virtual ~AbstractApple() { cout << "~AbstractApple\n"; }

};

// 中国苹果

class ChinaApple : public AbstractApple {

public:

  virtual void showName() { cout << "中国苹果\n"; }

  virtual ~ChinaApple() { cout << "~ChinaApple\n"; }

};

// 美国苹果

class USAApple : public AbstractApple {

public:

  virtual void showName() { cout << "美国苹果\n"; }

  virtual ~USAApple() { cout << "~USAApple\n"; }

};

// 日本

class JanpenApple : public AbstractApple {

public:

  virtual void showName() { cout << "日本苹果\n"; }

  virtual ~JanpenApple() { cout << "~JanpenApple\n"; }

};

// 抽象香蕉

class AbstractBanana {

public:

  virtual void showName() = 0;

  virtual ~AbstractBanana() { cout << "~AbstractBanana\n"; }

};

// 中国香蕉

class ChinaBanana : public AbstractBanana {

public:

  virtual void showName() { cout << "中国香蕉\n"; }

  virtual ~ChinaBanana() { cout << "~ChinaBanana\n"; }

};

// 美国香蕉

class USABanana : public AbstractBanana {

public:

  virtual void showName() { cout << "美国香蕉\n"; }

  virtual ~USABanana() { cout << "~USABanana\n"; }

};

// 日本香蕉

class JanpenBanana : public AbstractBanana {

public:

  void showName() { cout << "日本香蕉\n"; }

  virtual ~JanpenBanana() { cout << "~JanpenBanana\n"; }

};

// 抽象的梨

class AbstractPear {

public:

  virtual void showName() = 0;

  virtual ~AbstractPear() { cout << "~AbstractPear\n"; }

};

// 中国的梨

class ChinaPear : public AbstractPear {

public:

  virtual void showName() { cout << "中国的梨\n"; }

  virtual ~ChinaPear() { cout << "~ChinaPear\n"; }

};

// 美国的梨

class USAPear : public AbstractPear {

public:

  virtual void showName() { cout << "美国的梨\n"; }

  virtual ~USAPear() { cout << "~USAPear\n"; }

};

// 日本的梨

class JanpenPear : public AbstractPear {

public:

  virtual void showName() { cout << "日本的梨\n"; }

  virtual ~JanpenPear() { cout << "~JanpenPear\n"; }

};

然后是工厂的抽象，和具体的各种工厂类：

// 抽象工厂 针对产品族

class AbstrtactFactory {

public:

  virtual AbstractApple \*createApple() = 0;

  virtual AbstractBanana \*createBanana() = 0;

  virtual AbstractPear \*createPear() = 0;

};

// 中国工厂

class ChinaFatory : public AbstrtactFactory {

public:

  AbstractApple \*createApple() { return new ChinaApple(); }

  AbstractBanana \*createBanana() { return new ChinaBanana(); }

  AbstractPear \*createPear() { return new ChinaPear(); }

};

// 美国工厂

class UASFatory : public AbstrtactFactory {

public:

  AbstractApple \*createApple() { return new USAApple(); }

  AbstractBanana \*createBanana() { return new USABanana(); }

  AbstractPear \*createPear() { return new USAPear(); }

};

// 日本工厂

class JanpenFatory : public AbstrtactFactory {

public:

  AbstractApple \*createApple() { return new JanpenApple(); }

  AbstractBanana \*createBanana() { return new JanpenBanana(); }

  AbstractPear \*createPear() { return new JanpenPear(); }

};

调用如下：

int main(int argc, char const \*argv[]) {

  // 中国工厂生产的水果

  AbstrtactFactory \*cFac = new ChinaFatory();

  AbstractApple \*apple = cFac->createApple();

  AbstractBanana \*banana = cFac->createBanana();

  AbstractPear \*pear = cFac->createPear();

  apple->showName();

  banana->showName();

  pear->showName();

  delete apple;

  delete banana;

  delete pear;

  delete cFac;

  return 0;

}

输出：

中国苹果

中国香蕉

中国的梨

~ChinaApple

~AbstractApple

~ChinaBanana

~AbstractBanana

~ChinaPear

~AbstractPear

产品批次的类型组合的构成其实就是工厂方法的组合，而不论是哪个工厂生产的产品，是要是同一类的，就可以用抽象的产品上层来引用。

## 单例模式

只允许在程序中存在类的单个实例，分为懒汉式和饿汉式。

### 创建单例模式的必要的步骤：

* 把构造函数私有化private；
* 获取类实例时返回局部静态变量的类实例，或者是创建static静态的类成员变量（指针），用于指向创建的类实例；
* 调用public的接口获得静态的成员指针；

因为是把成员变量作为了静态的变量而且是单例的，所以单例模式不必考虑对象的释放问题。

### 懒汉式的单例模式：

// 单例模式演示类 懒汉式

class Singleton {

public:

  // 公有接口获取唯一实例

  static Singleton &getInstance() {

    cout << "获取实例" << endl;

    static Singleton instance;

    cout << "地址为:" << &instance << endl;

    return instance;

  }

private:

  // 私有构造函数

  Singleton() { cout << "构造函数启动。" << endl; };

  // 私有析构函数

  ~Singleton() { cout << "析构函数启动。" << endl; };

};

调用如下：

  cout << "main开始" << endl;

  thread t1([] { Singleton &s1 = Singleton::getInstance(); });

  thread t2([] { Singleton &s2 = Singleton::getInstance(); });

  t1.join();

  t2.join();

这里使用了两个线程来尝试获取单例的实例，输出如下：

main开始

获取实例

获取实例

构造函数启动。

地址为:地址为:0x7ff711bc50d0

0x7ff711bc50d0

这里的单例的创建是在genInstance函数中定义的局部静态变量，只有在手动第一次调用类Singleton::getInstance()方法时才会创建该单例的实例，第二次获取就不会再次创建了（局部静态变量的特点，只2初始化一次）。

通过输出也可以看出来，获取实例的动作有两次，但是实际调用构造函数只有一次。

注意：如果在，静态成员变量不是局部静态的，而是通过静态成员变量指针指向的，在多线程的场景下，对于new动作时没有原子性的。可能同时new出两个实例，这样就失去了单例的意义。所以用静态指针指向创建的类实例的懒汉式单例模式，是不具备线程安全特性的。

### 饿汉式的单例模式

如下：

//单例 饿汉模式

class SingletonHungery {

private:

  static SingletonHungery \*PSingleton;

  SingletonHungery() { cout << "SingletonHungery构造函数\n"; }

  ~SingletonHungery() { cout << "~SingletonHungery析构函数\n"; }

public:

  static SingletonHungery \*getInstance() { return PSingleton; }

};

SingletonHungery \*SingletonHungery::PSingleton = new SingletonHungery;

在饿汉使得单例模式中，单例类实例的创建并不是在内的成员函数中，而且获取类实例的共有接口getInstance也只是把静态的类实例成员变量(指针)返回而已。

static静态类实例的创建是在整个程序的静态区域，单独在全局域创建一次然后直接赋值给类的静态成员变量(指针).

调用如下：

  thread tt1([] { SingletonHungery \*s1 = SingletonHungery::getInstance(); });

  thread tt2([] { SingletonHungery \*s2 = SingletonHungery::getInstance(); });

  tt1.join();

  tt2.join();

  cout << "main结束" << endl;

输出：

SingletonHungery构造函数

main开始

main结束

因为是全局静态变量，所以饿汉式的单例在程序一开始运行就立即创建了，二不考虑师傅整的在程序中会使用到。从输出可以看出来，类实例的创建是早于main函数的。

而这里没有调用析构的原型是，全局静态变量伴随程序的整个生命周期，所以只要程序运行，它就不会析构，在程序自动退出后，会自动释放。

饿汉式的单例模式的优点是，类实例是全局静态的，只会在程序运行起来后初始化一次而且无论有多少个线程，都只有一次初始化，他是线程安全的。

## 代理模式

提供一种代理方式来控制对其他对象的访问。详细解释就是，在代理的类中，进行一系列的信息验证，只有通过验证的后，才能调用代理类提供的接口来访问目标对象。

如下是一个验证用户名和密码的代理类的例子：

class AbstratctInterface { // 这里抽象出一个接口类是为了能够让单例的同名run方法，替换访问目标对象的run

  virtual void run() = 0;

};

class MySystem : public AbstratctInterface {

public:

  MySystem() { cout << "MySystem \n"; }

  ~MySystem() {}

  void run() { cout << "系统启动...\n"; };

};

class SystemProxy : public AbstratctInterface {

private:

  MySystem \*sys;

  string userName;

  string password;

public:

  SystemProxy(string username, string password) {

    this->userName = username;

    this->password = password;

    this->sys = new MySystem; // 在代理内部初始化要访问的目标实例

  }

  ~SystemProxy() {

    if (this->sys) {

      delete this->sys;

    }

  }

  void run() {

    if (!checkAuth()) {

      cout << "验证未通过，权限不足!\n";

      return;

    }

    this->sys->run();

  }

  bool checkAuth() {

    if (this->userName != "admin" || this->password != "admin")

      return false;

    return true;

  }

};

注意到，这里先是定义了一个抽象类AbstratctInterface 用来限定代理和要访问的目标对象，拥有共同的访问接口，然后以便于在后续用代理类的同名进口替换访问目标的同名接口。

目标访问对象是创建在代理类中的。而且在代理类的实现过程中，增加了账户和密码检验的动作，在实际的场景中会结合更加具体的检验步骤。在调用代理类的run接口时，实际就是先经过代理层然后代理层校验通过之后，在进行对目标第一项的访问。

调用如下：

int main(int argc, char const \*argv[]) {

  SystemProxy \*sysproxy = new SystemProxy("root", "admin");

  sysproxy->run();

  /\* 输出：

  MySystem

  验证未通过，权限不足!

  \*/

  delete sysproxy;

  SystemProxy \*sysproxy2 = new SystemProxy("admin", "admin");

  sysproxy2->run();

  /\* 输出

  MySystem

  系统启动...

  \*/

delete sysproxy2;

  return 0;

}

输出：

MySystem

验证未通过，权限不足!

MySystem

系统启动...

第一次使用代理，没有传递正确的信息，未通过校验无法启动目标。第二次成功启动。

代理模式就是用新定义的代理类中的接口，来替换目标接口。然后在代理类中编制自己的检验规则用于控制独一目标的访问。在某些情况下， 一个对象不适合于或者不能直接应用拎一个对象，而代理对象可以做到在用户和目标对象之间起到中介的作用。

## 外观模式

根据迪米特法则，如果两个类不必互相通信，那么这两个累就不应发生联系。

外观模式是用于把不同的子系统，统一集成到一个类或者一个系统中，然后给用户提供一个一致的而又简单的交互界面。

下面是一个模拟智能家居通知屋子里面的电视，音响，灯的，使用外观模式的例子；

class Lock {

public:

  void On() { cout << "门锁打开\n"; }

  void Off() { cout << "门锁关闭\n"; }

};

class Light {

public:

  void On() { cout << "灯打开\n"; }

  void Off() { cout << "灯关闭\n"; }

};

class TV {

public:

  void On() { cout << "电视打开\n"; }

  void Off() { cout << "电视关闭\n"; }

};

class Audio {

public:

  void On() { cout << "音响打开\n"; }

  void Off() { cout << "音响关闭\n"; }

};

然后是Controller控制，模拟回家和离开家时的场景：

class Contoller {

private:

  Light \*light;

  TV \*tv;

  Audio \*audio;

  Lock \*lock;

public:

  Contoller() {

    light = new Light();

    tv = new TV();

    audio = new Audio();

    lock = new Lock();

  }

  ~Contoller() {

    delete light;

    delete tv;

    delete audio;

    delete lock;

  }

  void whenBackHome() {

    cout << "----------回家了-----------" << endl;

    lock->On();

    light->On();

    tv->On();

    audio->On();

  }

  void whenLeave() {

    cout << "--------出门了-------" << endl;

    lock->Off();

    light->Off();

    tv->Off();

    audio->Off();

  }

};

主调函数如下：

int main(int argc, char const \*argv[]) {

  Contoller \*ctrl = new Contoller();

  ctrl->whenBackHome();

  ctrl->whenLeave();

  delete ctrl;

  return 0;

}

输出：

----------回家了-----------

门锁打开

灯打开

电视打开

音响打开

--------出门了-------

门锁关闭

灯关闭

电视关闭

音响关闭

外观模式节省了用户和众多的子系统或者对象打交道的过程，通过“一键式”的简单调用，实现对众多的子系统的控制。

## 适配器模式

把一个类的接口转换成客户希望的另外一个类的接口，使得原本由于接口不兼容而不能一起工作的那些类能够一起工作。

其实适配器模式实现的方式有很多，常见的就是类的继承和组合，这一点在章节 [合成复用原则](#_合成复用原则) 有描述，如下是使用继承实现适配器的一个例子：  
一开始的主调函数如下：

  vector<int> v;

  for (int i = 0; i < 10; ++i) {

    v.push\_back(i);

  }

  for\_each(v.begin(), v.end(), MyPrint());

这里使用for\_each传递的第三个参数，在for\_each的使用的方法中，第三个参数是一个可调用的函数，用于把迭代的每一项传递进该函数，所以该函数是必须要有一个形参的。但是这时MyPrint()的实现如下：

// 在cpp中，struct可以当做类内使用，和class申明不同的是，struct中的成员默认是public的，

// class中只有添加了public的成员才是public

struct MyPrint { // 此处把struct当做类来使用，

  void operator()(int v1, int v2) { cout << v1 + v2 << endl; }

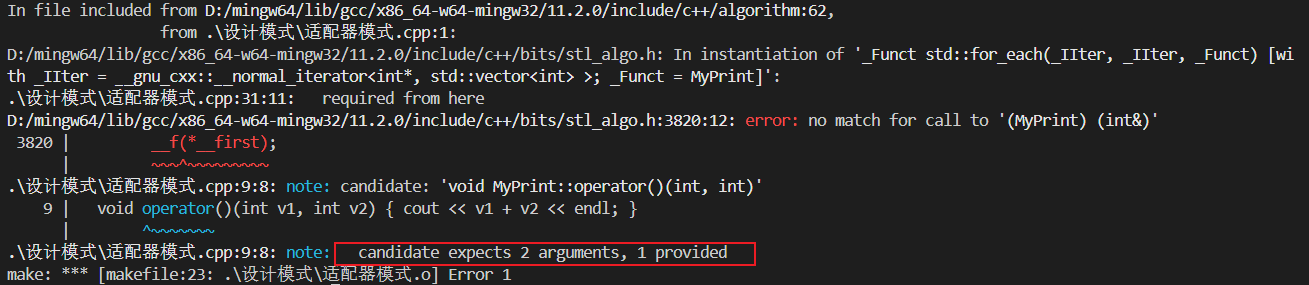
  // 这里直接定义()操作符，用于传递两个参数然后对两个参数进行加和计算

};

这里使用了struct构建一个结构体对象，在其中只包含一个重定义的operator()操作符，其功能是——该操作符有两个传参，然后把两个传参用于加和并输出。

这里关于struct和class的使用区别，struct默认内部成员是公开的，class只有在加了public: 声明的前提下，成员才是公开的。这里在MyPrint这个结构体中直接重定义了MyPrint的()操作符用于计算并打印两数加和，且默认公开（这也是一种cpp中函数式的一种非实现方式吧）。

此时MyPrint()创建的对象返回的计算方式函数需要两个参数，不符合for\_each的第三个参数对函数的要求，编译报错：



可以创建一个适配器，用于给for\_each传递第三个参数，如下：

class Adaptor : public MyPrint {

public:

  MyPrint print;

  int param;

  void operator()(int v) {

    print(v, param);

  } // 这里operator()的重定义，相当于给Adaptor定义了一个即时掉用方法，只要初始化Adaptor()经相当于调用和函数用于调用print(v,param)

  Adaptor(int param) {

    this->param = param;

  } // 通过构建是传递一个参数设置param,然后后续和operator()中传入的参数进行加和

};

在这个适配器Adaptor中，继承了MyPrint并重新实现了operator()的重定义，并且组合了MyPrint对象的用一个参数print保存住。

注意在这次的重定义中，是只有一个参数int v的，然后在操作符重定义的函数体中，调用从MyPrint继承过来的操作符()的方法，并向其传递内置的param参数和形参v。因为MyPrint类是被保存在print这个变量上的，所以直接print(v,param)就是调用的MyPrint的（）操作。

然后改写调用形式，因为此时Adaptor的操作符()的重定向，已经实现了传递单个参数的方式，另一个参与运算的参数param是在Adaptor构建是传递进去的。

for\_each(v.begin(), v.end(), Adaptor(100));

输出如下：

  100

  101

  102

  103

  104

  105

  106

  107

  108

  109

可以看到，在构建Adaptor是传递的poaram参数是100，而容器V中的每一项参数，和100加和并输出。实现了最终调用MyPrint的操作符()的调用。

## 模板方法模式

通过在一个新的类中，抽象定制一套工序步骤，然后将该套工序步骤用于集成的具体类去实现不同功能，并产出。

以冲泡茶叶和冲泡咖啡为例，不论哪种东西的冲泡，其步骤都是类似的，经历煮水，加料，倒水，加辅料的工序。把这些相似的工序抽象出来，作为一个抽象基类，然后具体的东西的制作继承自该抽象基类即可。

如下吗，先抽象出冲泡步骤：

using std::cout, std::endl;

class DrinkTemplate {   //抽象出冲泡步骤

public:

  virtual void BoildWater() = 0;

  virtual void AddTo() = 0;

  virtual void PourInCup() = 0;

  virtual void AddSomething() = 0;

};

然后具体的品种的工序继承自抽象基类：

class Tea : public DrinkTemplate {

public:

  virtual void BoildWater() { cout << "烧开水...\n"; }

  virtual void AddTo() { cout << "添加茶叶...\n"; }

  virtual void PourInCup() { cout << "向杯中倒入开水...\n"; }

  virtual void AddSomething() { cout << "加糖，做奶茶...\n"; }

  void make() {

    BoildWater();

    AddTo();

    PourInCup();

    AddSomething();

  }

};

class Coffee : public DrinkTemplate {

public:

  virtual void BoildWater() { cout << "烧开水...\n"; }

  virtual void AddTo() { cout << "添加咖啡...\n"; }

  virtual void PourInCup() { cout << "向杯中倒入开水...\n"; }

  virtual void AddSomething() { cout << "加糖，加奶...\n"; }

  void make() {

    BoildWater();

    AddTo();

    PourInCup();

    AddSomething();

  }

};

调用如下：

int main(int argc, char const \*argv[]) {

  Tea \*tea = new Tea();

  tea->make();

  cout<<"------------------------\n";

  Coffee \*coffee = new Coffee();

  coffee->make();

  delete tea;

  delete coffee;

  return 0;

}

输出：

烧开水...

添加茶叶...

向杯中倒入开水...

加糖，做奶茶...

------------------------

烧开水...

添加咖啡...

向杯中倒入开水...

加糖，加奶...

模板方法模式，可以理解为就是工序的抽象定制。他通过抽象出一些公共的工序步骤，然后定制接口用于约束后续的具体实现类去自定义实现这些公共步骤。

## 策略模式

策略模式，通过定义一系列的脱离客户端的独立的算法策略，然后用于客户自由适配，实现不同算法之间的相互替换。

如下是一个游戏中角色使用武器的例子，不同的武器就是不同的策略，角色charater可以自由更换和配置使用哪种武器，也就是更换哪种策略：

// 抽象武器策略

class WeaponStrategy {

public:

  virtual void UseWeapon() = 0;

};

class Knife : public WeaponStrategy {

public:

  virtual void UseWeapon() { cout << "使用匕首\n"; }

};

class AK47 : public WeaponStrategy {

public:

  virtual void UseWeapon() { cout << "使用AK47\n"; }

};

class Character {

private:

  WeaponStrategy \*pWeapon;

public:

  ~Character() { delete pWeapon; }

  void setWeapon(WeaponStrategy \*weapon) { // 角色设置武器

    this->pWeapon = weapon;

  }

  void brandishWeapon() { // 角色使用户武器

    this->pWeapon->UseWeapon();

  }

};

调用如下：

int main(int argc, char const \*argv[]) {

  Character \*c = new Character();

  Knife \*knife = new Knife();

  AK47 \*ak = new AK47();

  c->setWeapon(knife); // 配置策略

  c->brandishWeapon();

  c->setWeapon(ak); // 更换策略

  c->brandishWeapon();

  delete c;

  delete knife;

  delete ak;

  return 0;

}

输出：

使用匕首

使用AK47

创建的角色和创建的武器（策略），都是互相独立的，角色可以随时更换策略。

## 命令模式

把请求封装为一个个的类，请求的所有具体细节都将在请求类中实现，客户端要想服务端发送请求时，只需要简单的调用即可。而且通过封装的一个个请求，服务端可以把请求放入到待处理队列中，或者是对一个个的请求进行延迟，撤销等操作。

如下是以游戏场景中，对玩家的升级，添加金币，添加钻石等等的请求，每一个都封装为一个单独的类，然后在服务端类中，可以把这些发送的请求的类实例添加到服务端的队列中，然后依次处理：

先是服务端各种命令的处理接口/协议：

// 处理客户端各种命令的方式

class HandleClientProtocal {

public:

  void AddMoney() { cout << "给玩家增加金币\n"; }

  void AddDiamond() { cout << "给玩家增加钻石\n"; }

  void AddEquipment() { cout << "给玩家增加装备\n"; }

  void addLevel() { cout << "给玩家升级\n"; }

};

然后是抽象出来命令类的抽象基类：

class AbstractCommand {

public:

  virtual void handle() = 0;

};

然后是具体的各种命令，每一种对应一个继承自命令抽象基类的子类：

// 给玩家增加钱币的请求

class AddMoneyComamnd : public AbstractCommand {

private:

  HandleClientProtocal \*pProtocal;

public:

  AddMoneyComamnd(HandleClientProtocal \*protocal) {

    this->pProtocal = protocal;

  }

  virtual void handle() { this->pProtocal->AddMoney(); }

};

// 给玩家增加钻石的请求

class AddDiamondCommand : public AbstractCommand {

private:

  HandleClientProtocal \*pProtocal;

public:

  AddDiamondCommand(HandleClientProtocal \*protocal) {

    this->pProtocal = protocal;

  }

  virtual void handle() { this->pProtocal->AddDiamond(); }

};

// 给玩家增加装备的请求

class AddEquipmentCommand : public AbstractCommand {

private:

  HandleClientProtocal \*pProtocal;

public:

  AddEquipmentCommand(HandleClientProtocal \*protocal) {

    this->pProtocal = protocal;

  }

  virtual void handle() { this->pProtocal->AddEquipment(); }

};

// 给玩家升级的请求

class AddLevelCommand : public AbstractCommand {

private:

  HandleClientProtocal \*pProtocal;

public:

  AddLevelCommand(HandleClientProtocal \*protocal) {

    this->pProtocal = protocal;

  }

  virtual void handle() { this->pProtocal->addLevel(); }

};

每一个命令类都实现命令抽象基类的接口handle()，但是具体handle是如何处理的不在命令类中实现，也就是说命令在服务端是如何实现的，命令类并不关心，命令类只封装请求，具体服务端的处理方法是在一开始的协议中实现的。

然后是服务端，这里通过在内定义一个命令类的队列，可以把模拟传来的客户端的各种命令添加到队列中，然后迭代处理：

// 服务器端

class Server {

public:

  queue<AbstractCommand \*> qCommands;

  void addRequest(AbstractCommand \*cmd) { this->qCommands.push(cmd); }

  void startHandle() {

    while (!qCommands.empty()) {

      Sleep(1000);

      AbstractCommand \*cmd = qCommands.front();

      cmd->handle();

      qCommands.pop();

    }

  }

};

主调函数如下：

int main(int argc, char const \*argv[]) {

  // 创建处理客户端的处理协议的类实例

  HandleClientProtocal \*handleProtocal = new HandleClientProtocal();

  // 客户端的增加金币的请求

  AddMoneyComamnd \*addMoney = new AddMoneyComamnd(handleProtocal);

  // 客户端增加钻石的请求

  AddDiamondCommand \*addDiamond = new AddDiamondCommand(handleProtocal);

  // 客户端增加装备的请求

  AddEquipmentCommand \*addEquipment = new AddEquipmentCommand(handleProtocal);

  // 客户端升级等级的请求

  AddLevelCommand \*addLevelCommand = new AddLevelCommand(handleProtocal);

  Server \*server = new Server();

  server->addRequest(addMoney); // 把各个请求加入到服务端的处理队列中

  server->addRequest(addDiamond);

  server->addRequest(addEquipment);

  server->addRequest(addLevelCommand);

  // 服务器开始处理请求

  server->startHandle();

  return 0;

}

在主调函数中，通过创建一个个的命令类的实例，然后让服务器类Server的实例，把这些创建的命令类的实例计入到自己的队列中，调用stratHandle迭代处理，输出如下：

给玩家增加金币

给玩家增加钻石

非玩家增加装备

给玩家升级

命令模式的优势在于其边界的扩展性，比如在本例中要新增一个增加一个给玩家提升PH血量的方法，那么只需要在一开始的协议中增加相应的协议方式（也可以使用子类继承或者组合原协议），然后再增加一个对应的命令类即可，而在Server类中，无需任何的修改。

## 观察者模式

观察者模式中分为蒸菜观察的观察者，和被观察的对象。当被观察的对象发生某种变化的时候，它可以通知众多的观察者，以便观察者做出相应的动作。

以游戏中的场景为例，当有多个角色在攻击关卡中的BOSS的时候，当BOSS死亡时，玩家都停止攻击。这里的BOSS就是被观察者，而玩家就是观察者。

为了定制丰富的观察者角色和被观察的角色，可以对二者都进行上层抽象，但是重要的是被观察对象需要有通知观察者的接口。

如下是模拟首相的英雄角色和具体的角色英雄：

// 抽象的英雄角色

class AbstratctHero {

public:

  virtual void Update() = 0;

};

class HeroA : public AbstratctHero {

public:

  HeroA() { cout << "英雄A正在攻击...\n"; }

  virtual void Update() { cout << "英雄A停止攻击，等待状态...\n"; }

};

class HeroB : public AbstratctHero {

public:

  HeroB() { cout << "英雄B正在攻击...\n"; }

  virtual void Update() { cout << "英雄B停止攻击，等待状态...\n"; }

};

class HeroC : public AbstratctHero {

public:

  HeroC() { cout << "英雄C正在攻击...\n"; }

  virtual void Update() { cout << "英雄C停止攻击，等待状态...\n"; }

};

然后是抽象的关卡BOSS和具体的关卡BOSSA:

// 抽象的关卡BOSS

class AbstractBOSS {

public:

  virtual void recvChallenge(AbstratctHero \*hero) = 0;

  virtual void deleteChallenge(AbstratctHero \*hero) = 0;

  virtual void updateChallenge() = 0;

};

class BOSSA : public AbstractBOSS {

private:

  std::list<AbstratctHero \*> challengeList;

public:

  BOSSA() {}

  virtual void recvChallenge(AbstratctHero \*hero) {

    cout << "BOSSA 接受挑战...\n";

    challengeList.push\_back(hero);

  }

  virtual void deleteChallenge(AbstratctHero \*hero) {

    challengeList.remove(hero);

  }

  virtual void updateChallenge() {

    for (std::list<AbstratctHero \*>::iterator it = challengeList.begin();

         it != challengeList.end(); it++) {

      (\*it)->Update();

    }

  }

};

// 还可以实现其他的关卡BOSS...

注意到这里的实际的被观察对象BOSSA，其内部是有一个list<AbstractHero\*>的，用于存放当前有哪些观察者在观察BOSSA这个对象，这很重要，因为通知观察者的时候，需要这个结构。

如下是调用函数中模拟英雄角色攻击关卡BOSS的场景：

int main(int argc, char const \*argv[]) {

  HeroA \*h1 = new HeroA();

  HeroB \*h2 = new HeroB();

  HeroC \*h3 = new HeroC();

  BOSSA \*boss = new BOSSA();

  boss->recvChallenge(h1);

  boss->recvChallenge(h2);

  boss->recvChallenge(h3);

  boss->deleteChallenge(h1);

  cout << "英雄A死亡" << endl;

  cout << "BOSSA 死亡..." << endl;

  boss->updateChallenge(); // 通知所有的观察者更新状态

  return 0;

}

输出：

英雄A正在攻击...

英雄B正在攻击...

英雄C正在攻击...

BOSSA 接受挑战...

BOSSA 接受挑战...

BOSSA 接受挑战...

英雄A死亡

BOSSA 死亡...

英雄B停止攻击，等待状态...

英雄C停止攻击，等待状态...

可以看到当观察者英雄角色在取消观察后，随后当被观察对象的关卡BOSS死亡以后，正确通知了其余英雄角色。

# 智能指针

## 独占型智能指针unique\_ptr

特点：

* 脱离代码块或者是作用域后，自动调用类型的析构函数;
* 把原unique\_ptr指针变量重新赋值给一个新创建的智能指针，原指针将调用类型的析构函数；
* 不能用同类型的新定义的unique\_ptr智能指针变量，给其使用已有的unique\_ptr指针对其直接进行赋值， 需要使用move移动语义才能进行赋值。

定义了智能指针后，可以通过调用.get()获取智能指针包装的内存区域的地址。

下面对比普通指针和智能指针的差异，从而认识到智能指针的优势。

class A {

public:

  explicit A(int n);

  A();

  A(const A &a);

  ~A();

  void setNum(int v);

  void info() const; // 后置const限定函数内不能修改成员值

private:

  int num = 0;

};

A::A(int n) : num(n) {

  std::cout << "A的带参构造函数,num=" << num << std::endl;

}

A::A() { std::cout << "A的默认构造函数，num=" << num << std::endl; }

A::A(const A &a) { "A的拷贝构造函数\n"; }

A::~A() { std::cout << "A的析构函数,num=" << num << std::endl; }

void A::info() const { std::cout << "info信息,num=" << num << std::endl; }

void A::setNum(int v) { num += v; }

这里定义了一个类A,分别实现了其五参数的构造函数，带参构造函数和拷贝构造函数，以及几个成员函数。

然后调用如下：

  // 栈上

  A a\_arg(1); // 栈上创建的类实例化对象，会在函数retuen前会自动调用析构

  A a\_noarg(); // 无参的构造函数可能会被优化

  // 堆上

  int \*p1 = new int(100);

  A \*a = new A(111); // 堆上new出的实例化对象，如果不调用delete会有内存泄漏

  {

    delete p1; // 如果要重用指针，需要在重用前delete旧的指向内存

    p1 = new int(200); // p1被重新使用，二次new改变了指向

    // delete p1;  【非法】在重用后delete，会导致之前p1指向的地址内存泄漏

    // 重用了外部的堆上的指针p1，但是后续没有delete,会有内存泄漏问题

    A(2); // 块中的栈实例化也会自动调用析构

  }

  std::cout << \*p1 << std::endl;

  delete p1;

这里分别在栈上和堆上创建A的实例化对象。栈上创建的内存会自动调用析构函数销毁没什么问题。但是在堆上创建的内存，使用p1指向后，随后在内部代码块中重用，如果按照注释中的做法把delete语句放在重用之后，那么就会存在着第一次申请的内存丢失导致的内存泄漏，因为在重用更改指针的指向后，第一次申请的内存区域将无法访问， 而delete释放的内存是第二次指向的，第一次指向的内存此时无法通过任何途径的能到地址，无从释放。

而且这里还存在着二次释放的问题，比如如果按照代码块中注释的做法，在指针重用后delete p1，然后又在代码块外部再次delete p1,虽然此时delete和new 操作是成对出现的，但是实际上两次delete都是对第二次new的操作，而第一次new的内存此时已丢失，造成内存泄漏。

这种导致内存泄漏的核心原因是，在重用指针之前，需要使用原指针把原来申请的内存释放，一旦指针重重用了但是之前的指向的内存没能得到释放，再重新指向后原指向就会丢失，后续无法访问也无法释放。

智能指针unique\_ptr智能的地方便是在于，使用了智能指针包装了指针后，一旦指针被重用，断开了原来的指向关系，那么智能指针会自动调用类型的析构函数，而无须担心析构问题。

### 智能指针unique\_ptr的创建和使用

#### unique\_ptr的使用方式1——用新定义过的智能指针对原始指针进行”包装”：

  cout << "------------智能指针使用 start---------------\n";

  /\* \*\*\*\*\*\*智能指针使用方式1：把comm\_a包装为智能指针ua \*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  A \*comm\_a = new A(1000);

  std::unique\_ptr<A> ua(comm\_a);

  cout << "delete 原始指针前，原始指针和智能指针都可以访问" << endl;

  comm\_a->info();

  ua->info();

  // delete comm\_a;  //原始指针赋值为空即可，不可进行delete

  comm\_a = nullptr;

  cout << "delete原始指针后,智能指针将不可以再用" << endl;

  // ua->info();

  cout << "delete原始指针后,原始指针不可以再使用" << endl;

  // comm\_a->info();

输出：

------------智能指针使用 start---------------

A的带参构造函数,num=1000

delete 原始指针前，原始指针和智能指针都可以访问

info信息,num=1000

info信息,num=1000

delete原始指针后,智能指针将不可以再用

delete原始指针后,原始指针不可以再使用

这里同时描述了一个使用unique\_ptr时候，在使用该智能指针包装原始指针之后，不能对原始指针进行delete,因为本质上智能指针是在原始指针的基础上多了一层“包装“，但是实际的对象内存还是同一份，如果对原始指针调用delete将会导致智能指针也变得不可访问。

#### unique\_ptr智能指针的使用方式2——新定义智能指针后直接用new出的对象初始化：

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 智能指针使用方式2:直接new类型构造给unique\_ptr \*/

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* std::unique\_ptr<A> ua2(new A(1000)); \*\*\*\*\*\*\*/

  /\* 或者 \*/ std::unique\_ptr<A> ua2{new A(1001)};

  ua2->info();

  cout << "自定义类A 智能指针地址ua2" << ua2.get() << endl;

  std::unique\_ptr<int> uaint2{new int(10)};

  cout << "常规类型int 智能指针地址uaint2:" << uaint2.get() << endl;

  /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\* 方式2 end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

输出：

A的带参构造函数,num=1001

info信息,num=1001

自定义类A 智能指针地址ua2 0x28454ef2200

常规类型int 智能指针地址uaint2: 0x28454ef2220

A的带参构造函数,num=1002

自定义类A 智能指针地址ua30x28454ef2200

常规类型int 智能指针地址uaint3:0x28454ef2260

这里分别用自定义类A和普通int类型举例。从输出可以看出来，在使用new初始化智能指针的时候，类型的构造函数被自动调用，完成后，可以用整你指针使用->操作符访问类成员函数，使用.get()获得智能指针中包含的对象的指针等。

#### unique\_ptr智能指针的方式3——使用std::make\_uniqu<T>创建类型T的整你指针

  std::unique\_ptr<A> ua3 = std::make\_unique<A>(1002);

  cout << "自定义类A 智能指针地址ua3" << ua2.get() << endl;

  std::unique\_ptr<int> uaint3 = std::make\_unique<int>(12);

  cout << "常规类型int 智能指针地址uaint3:" << uaint3.get() << endl;

输出：

A的带参构造函数,num=1002

自定义类A 智能指针地址ua30x28454ef2200

常规类型int 智能指针地址uaint3:0x28454ef2260

因为还没有到达调用函数的结束位置，所以，类的硒鼓还未被自动调用。确定的一点是，使用智能指针后，在代码块结束时，类对象的析构函数会自动调用的。

#### 验证unique\_ptr离开作用域时的自动析构

然后观察在内部代码块中，智能指针的表现，：

  {

    cout << "-------------内部代码块 start-------------\n";

    A \*inner\_a = new A(111);

    std::unique\_ptr<A> inner\_ua(inner\_a); // 方式1

    inner\_a->info();

    int \*inner\_i = new int(11);

    std::unique\_ptr<int> inner\_uinti(inner\_i); // 方式1

    std::unique\_ptr<A> inner\_uanew(new A(112)); // 方式2

    inner\_uanew->info();

    std::unique\_ptr<int> inner\_uintnew(new int(12)); // 方式2

    std::unique\_ptr<A> inner\_makea = std::make\_unique<A>(113); // 方式3

    inner\_makea->info();

    std::unique\_ptr<int> inner\_makeint = std::make\_unique<int>(13); // 方式3

  }

  cout << "-------------内部代码块 end-------------\n";

输出：

-------------内部代码块 start-------------

A的带参构造函数,num=111

info信息,num=111

A的带参构造函数,num=112

info信息,num=112

A的带参构造函数,num=113

info信息,num=113

A的析构函数,num=113

A的析构函数,num=112

A的析构函数,num=111

-------------内部代码块 end-------------

从输出可以看出，无论使用哪种方式创建的智能指针，在代码块结束前会地自动调用类型的析构函数，这样就正确了手动添加delete的步骤，而且可以避免delete删除的时机不对导致的各种问题。

### Unique\_ptr的传递

智能指针也有值传递和应用传递两种方式

一般都是出现在函数调用过程中，如下定义了的一些函数，其中func形参是unique\_ptr指针值传递类型的，函数func2是引用类型的，func3是限定const形参的类型：

using std::unique\_ptr;

/\* 智能指针的指针值传递 \*/

void func(unique\_ptr<A> a) { a->info(); }   // 指针值传递

void func2(unique\_ptr<A> &a) { a->info(); } // 指针引用传递

void func3(const unique\_ptr<A> &a) {        // 指针引用传递

  a->setNum(10);

  a->info();

}

#### unique\_ptr的指针值传递

下面进行unique\_ptr指针值传递调用时发生的事：

  cout << "-----------------智能指针的指针值传递 start---------------\n";

  unique\_ptr<A> uniq\_a = std::make\_unique<A>(50);

  // func(uniq\_a);  //错误，智能指针传递指针值时需要移动语义后再进行传递

  cout << "\*\*\*\*\*智能指针值传递move，传递之前\*\*\*\*\*\n";

  func(std::move(uniq\_a));

  cout << "\*\*\*\*\*智能指针值传递move，传递之后\*\*\*\*\*\n";

  // uniq\_a->info();   //错误，智能指针在move被传递后，会被析构，将不可再访问

输出：

-----------------智能指针的指针值传递 start---------------

A的带参构造函数,num=50

\*\*\*\*\*智能指针值传递move，传递之前\*\*\*\*\*

info信息,num=50

A的析构函数,num=50

\*\*\*\*\*智能指针值传递move，传递之后\*\*\*\*\*

这里先创建了一个A类的智能智能指针的实例，然后尝试直接调用func按照传递值的方式传递智能指针的值，但是会有报错，因为unique\_ptr的特点是：指向关系固定不能随意更改，且指向的内存区域也不能直接用其他同类型的副本指针去二次指向，也就是不用用两个智能指针指向同一个内存对象。因为在调用函数进行值传递的过程中会创建指针副本，这就和unique\_ptr是违背的，所以会有语法错误。

要想完全使用值传递方式来传递unique\_ptr，就需要使用std::move移动语义来转换智能指针所指向的内存的所有权。这里在正式调用func时，使用move(uniq\_a)对智能指针进行了所有权的转换，那么在函数func调用时，所有权就被传递到func函数的参数a上。

不过这里需要注意一点，在所有权转让到函数参数a之后，随着func函数的结束，而又没有进行指针的返回，那么在函数func调用完毕回到主调函数时，func中的a智能指针会自动调用类型对象的析构函数，所以在func返回主函数后，uniq\_a指向的对象将变得不可访问，因为对象在func调用完毕后就被析构了。

如下在调用指针值传递函数时，传参时直接初始化智能指针，这样的传递方式，将会自动把智能指针进行move.如下调用：

  cout << "\*\*\*\*\*智能指针在传递时默认带有move移动语义\*\*\*\*\*\n";

  func(std::make\_unique<A>());

  func(std::make\_unique<A>(51));

  cout << "-----------------智能指针的指针值传递 end---------------\n";

输出：

\*\*\*\*\*智能指针在传递时默认带有move移动语义\*\*\*\*\*

A的默认构造函数，num=0

info信息,num=0

A的析构函数,num=0

A的带参构造函数,num=51

info信息,num=51

A的析构函数,num=51

-----------------智能指针的指针值传递 end---------------

从输出可以看出，即使new出的对象是被传递到func函数中的，因为在func函数中正常调用了对象的方法。而且在调用后，创建的对象也被自动析构了。

#### unique\_ptr的引用传递

func2函数是引用传递的：

void func2(unique\_ptr<A> &a) { a->info(); } // 指针引用传递

如下代码验证引用传递：

  cout << "-----------------智能指针的指针引用传递 start---------------\n";

  unique\_ptr<A> uniq\_ca = std::make\_unique<A>(52);

  func2(uniq\_ca);

  uniq\_ca->info(); // 智能指针引用传递结束后，智能指针不会被析构，还可以访问

  func3(uniq\_ca);

  uniq\_ca->info();

  cout << "-----------------智能指针的指针引用传递 end---------------\n";

输出：

-----------------智能指针的指针引用传递 start---------------

A的带参构造函数,num=52

info信息,num=52

info信息,num=52

info信息,num=62

info信息,num=62

-----------------智能指针的指针引用传递 end---------------

使用引用传递，因为没有调用move的所有权转让的问题，因此在func2调用返回后，智能指针uniq\_ca在主调函数中仍然能够正常访问。

所以，如果是想要保持主调函数中的智能指针生命一直有效。就使用引用传递方式调用函数。

#### 不能同类型的unique\_ptr智能指针直接指向已有的智能指针

Unique\_ptr有其独占性，不能直接用第二指针个变量直接再次指向旧的智能指针。如果场景必须要对智能指针重新赋值，那么需要使用移动语义std::move对旧的智能指针转让内存所有权。

  cout << "---------不能用同类型的新智能指针直接指向源智能指针 start----------\n";

  unique\_ptr<A> ua\_init(new A(111));

  // unique\_ptr<A> another\_ua = ua\_init;  错误

  cout << "\*\*\*\*\*需要使用移动语义move对已有的智能指针更换所有权到新智能指针\*\*\*\*\*\n";

  unique\_ptr<A> ua\_will\_move(new A(112));

  cout << "move之前，源数据:";

  ua\_will\_move->info();

  unique\_ptr<A> ua\_moved = std::move(ua\_will\_move);

  cout << "move之后，目标数据:";

  ua\_moved->info();

  cout << "---------不能用同类型的新智能指针直接指向源智能指针 end----------\n";

输出：

---------不能用同类型的新智能指针直接指向源智能指针 start----------

A的带参构造函数,num=111

\*\*\*\*\*需要使用移动语义move对已有的智能指针更换所有权到新智能指针\*\*\*\*\*

A的带参构造函数,num=112

move之前，源数据:info信息,num=112

move之后，目标数据:info信息,num=112

---------不能用同类型的新智能指针直接指向源智能指针 end----------

在这段代码中，先创建了一个A类型的智能指针ua\_init，指向了new出来的A类对象。然后尝试直接用ua\_init去给新定义的同类型智能指针another\_ua赋值，本意是创建一个指针副本，但是这在unique\_ptr是不允许的，会有语法错误。

如果需要一个新的同类型指针来指向历史创建的unique\_ptr智能指针所代表的对象，那么必须使用std::move移动语义把历史的智能指针的所有权转让给新的指针才行。这里重新创建了一个ua\_will\_move的智能指针指定new出来的A类对象，然后用move移动ua\_will\_move的所有权给同类型的智能指针ua\_moved。可以看到move后，使用新的ua\_moved指针就可以正常访问对象方法了。

#### unqie\_ptr对象的所有权切断

在unique\_prte对对象的所有权被切断后，且没有新的指针能够替换指向(所有权)，那么对象将自动调用析构函数：

  cout << "---------智能指针被重新复制后，原数据将调用析构 start--------\n";

  ua\_init->info();

  ua\_init = unique\_ptr<A>(new A(222));

  ua\_init->info();

  cout << "---------智能指针被重新复制后，原数据将调用析构 end--------\n";

输出：

---------智能指针被重新复制后，原数据将调用析构 start--------

info信息,num=111

A的带参构造函数,num=222

A的析构函数,num=111

info信息,num=222

---------智能指针被重新复制后，原数据将调用析构 end--------

这里的ua\_init沿用上一节创建的unique\_ptr智能指针。从输出可以看出来，我们让ua\_init重新指向了一个新创建的同类型智能指针，其背后是一个新new出来的A对象，而原来的ua\_init指向的A类对象num=111自动调用了析构函数。也就是对智能指针ua\_init的复用指向新创建的智能指针对象后，原来的所有权关系被切断，原智能指针建立了新的所有权关系，那么原类型的对象就会自动调用析构函数。

### 返回函数中创建的unque\_ptr变量

如下是自定义的类A:

/\* 类A \*/

class A {

public:

  explicit A(int n);

  A();

  A(const A &a);

  ~A();

  void setNum(int v);

  void info() const; // 后置const限定函数内不能修改成员值

private:

  int num = 0;

};

A::A(int n) : num(n) {

  std::cout << "A的带参构造函数,num=" << num << std::endl;

}

A::A() { std::cout << "A的默认构造函数，num=" << num << std::endl; }

A::A(const A &a) { "A的拷贝构造函数\n"; }

A::~A() { std::cout << "A的析构函数,num=" << num << std::endl; }

void A::info() const { std::cout << "info信息,num=" << num << std::endl; }

void A::setNum(int v) { num += v; }

/\* 类A end \*/

如下函数get\_uniq\_ptr，在该函数中，创建了一个unique\_ptr的智能指针呢，然后将其从函数return：

/\* 函数中创建的智能指针return出函数 \*/

unique\_ptr<A> get\_uniq\_ptr(int v = 0) {

  unique\_ptr<A> a;

  if (v)

    a = make\_unique<A>(v);

  else

    a = make\_unique<A>();

  cout << "address a.get():" << a.get()

       << endl;                        // address a.get():0x187c37521e0

  cout << "address &a:" << &a << endl; // address &a:0x4e587ffca8

  /\* 智能指针的.get()得到的地址和直接&得到的地址不同 \*/

  return a; // 返回函数内创建的智能指针

}

在这个函数中，判断形参v是否传递了参数，如果a不为默认值0，那么就使用v去初始化智能指针new出的类实例。然后在查看创建的智能指针对应的对象的地址和智能指针本身的地址。

主调函数中，调用如下：

  unique\_ptr<A> a\_in\_main = get\_uniq\_ptr();

  a\_in\_main->info(); // 可以正常访问在函数中创建的智能指针

  get\_uniq\_ptr(11)->info(); // 直接调用时可以链式访问函数内创建的智能指针

输出：

A的默认构造函数，num=0

address a.get():0x21ec6f92210

address &a:0xe09a7ffb48

info信息,num=0

A的带参构造函数,num=11

address a.get():0x21ec6f92230

address &a:0xe09a7ffb50

info信息,num=11

A的析构函数,num=11

从调用中可以看出来，调用函数get\_uniq\_ptr()得到了函数中创建的智能指针，赋值给a\_in\_mian的智能指针后，在主调函数中仍然能够正常访问函数中创建的对象的info方法。

第二次调用使用了链式的访问成员函数，因为在主调函数中没有任何变量对返回的智能指针的指向，在调用结束后，A的实例对象num=11也就即时调用了析构函数，从最后一句打印可以看出，调用完info后对象直接析构了。

##### 在被调用函数中修改了unique\_ptr的指向

如下函数change\_direction，该函数的参数是unique\_ptr的引用类型，但是在函数中对应用的智能指针执行了新的new出的对象(从函数get\_uniq\_ptr返回的)：

void change\_direction(/\* const \*/ unique\_ptr<A> &a) {

  cout << "change\_direction\n";

  a = get\_uniq\_ptr(); // 如果const 修饰形参后，就不可以改变源数据的指向

  //这里a被重新赋值指向另外的对象，源数据a背后的析构函数将会被调用

  return;

}

由于智能指针指向的改变，也就意味着原来的所有权关系的切断，那么传递进来的a背后原来的实例对象将调用析构函数，而因为是应用传递，新创建的指向关系将在主调函数中生效，在函数结束调用后，在主调函数中a代表的将会是在函数中创建的新的实例，

主调函数如下：

  unique\_ptr<A> aa = make\_unique<A>(100);

  change\_direction(aa);

  cout << "change\_direction end.\n";

  // 在调用的函数中修改了智能指针的指向，源数据aa的析构函数将被调用

  aa->info();   //将访问的是新指向的数据中的

  cout<<"---------main end ----------\n";

  return 0;

输出：

A的带参构造函数,num=100

cannt\_change\_direction

A的默认构造函数，num=0

address a.get():0x21ec6f92250

address &a:0xe09a7ffa98

A的析构函数,num=100

cannt\_change\_direction end.

info信息,num=0

---------main end ----------

A的析构函数,num=0

A的析构函数,num=0

从输出可以看出，A对象num=100是在主调函数中创建的，但是在传递到change\_direction函数中后，因为指向关系被切断，对象num=100的析构函数被调用。同时在函数中，新new出来的默认参数对象num=0被切换给了主调函数中的aa对象，所以在主调函数中访问info函数，打印的是新的对象的内容num=0(在函数中创建的无参的A实例对象)。

而最后打印的析构信息是在输掉函数结束后，自动对未释放的对象调用析构函数。

## 带引用计数的智能指针shared\_ptr

智能指针shared\_ptr是一个内部有引用计数的智能指针;

* 在离开定义shared\_ptr的代码块或者作用域时，而又没有进行return返回出来，指针将调用类型的析构；
* 不同于unique\_ptr, shared\_ptr可以定义同类型的指针变量，使用已有的shared\_ptr对其进行赋值，相当于可以新增多份指针对同一份内存对象进行指向；同时引用计数加1；
* 对同一个数据源的智能指针，如果某个shared\_ptr赋值为nullptr，那么引用计数减1；
* 当引用计数减小到0时，自动调用类型的析构函数；

.use\_count()接口可以查看当前的引用数目。

### 智能指针shared\_ptr的创建和使用

#### shared\_ptr使用方式1——包装原始指针

这个和uniqe\_ptr的做法一样，指针名换成shred\_ptr即可，下面重点介绍方式2和方式3

#### shared\_ptr使用方式2——直接new类型，构造给shared\_ptr

如下分别是构建普通类型int和自定义类型A，然后使用shared\_ptr的方式：

  shared\_ptr<int> sint\_newptr = shared\_ptr<int>{new int(10)};

  cout << \*sint\_newptr << " use\_count:" << sint\_newptr.use\_count() << endl;

  shared\_ptr<A> sa\_newptr = shared\_ptr<A>{new A(100)};

  sa\_newptr->info();

  cout << " use\_count:" << sa\_newptr.use\_count() << endl;

输出：

10 use\_count:1

A的带参构造函数,num=100

info信息,num=100

use\_count:1

对于普通类型如int，这里直接使用\*取内容即可得到shared\_ptr智能指针背后的对昂的内容。

通过输出可以看出来，一旦shared\_ptr被构建出来，引用计数就是1.

#### shared\_ptr使用方式3——使用std::make\_shared方法

如下是分别对原始类型int和自定义类型A，使用shared\_ptr构建智能指针：

  shared\_ptr<int> sint\_makeptr = make\_shared<int>(11);

  cout << \*sint\_makeptr << endl;

  cout << " use\_count" << sint\_makeptr.use\_count() << endl;

  shared\_ptr<A> sa\_makeptr = make\_shared<A>(12);

  sa\_makeptr->info();

  cout << " use count:" << sa\_makeptr.use\_count() << endl;

输出：

11 use\_count1

A的带参构造函数,num=12

info信息,num=12

use count:1

#### 使用已有的shared\_ptr给新创建的shared\_ptr赋值

使用已有的shared\_ptr给新创建的shared\_ptr赋值，会增加引用计数。其实可以这么理解，shared\_ptr就是对象的“标签“，初始创建时，只有一个shred\_ptr也就是一个标签，然后新定义了同类型的shared\_ptr标签后，用已有的标签给新的标签设定了指向同一个对象，那么对象就会多一个标签，这个次数就是引用计数。

如下代码演示的是，使用已有的指针对新定义的shared\_ptr指针进行赋值，然后查看指针的引用计数：

  shared\_ptr<int> sint\_newptr2 = sint\_newptr;

  shared\_ptr<A> sa\_newptr2 = sa\_newptr;

  cout << "sint\_newptr2.use\_count: " << sint\_newptr2.use\_count()

       << " sint\_newptr2.use\_count " << sa\_newptr2.use\_count() << endl;

  shared\_ptr<int> sint\_makeptr2 = sint\_makeptr;

  shared\_ptr<A> sa\_makeptr2 = sa\_makeptr;

  cout << "sint\_makeptr2.use\_count: " << sint\_makeptr2.use\_count()

       << " sa\_makeptr2.use\_count " << sa\_makeptr2.use\_count() << endl;

输出：

sint\_newptr2.use\_count: 2 sint\_newptr2.use\_count 2

sint\_makeptr2.use\_count: 2 sa\_makeptr2.use\_count 2

从输出可以看出来，sint\_newptr2和sa\_newptr以及 sin\_makeptr2和sa\_makeptr2的应用计数use\_count都是2，因为他们都是用已有的shared\_ptr进行的赋值的。

使用已有的shared\_ptr给新的shared\_ptr赋值，相当于增加了一个指针指向，但实际上的数据还是同一份，只要通过新旧任意一个指针重新给变量的操作，那么也会反映到另外一个指针上来。

如下使用智能指针，修改圆数据的值：

  \*sint\_newptr2 \*= 10;

  cout << \*sint\_newptr << " " << \*sint\_newptr2 << endl;

  \*sint\_makeptr \*= 10;

  cout << \*sint\_makeptr2 << " " << \*sint\_makeptr << endl;

输出：

100 100

110 110

这里通过新指针给源数据在原来的基础上\*10后，通过新旧任意一个shared\_ptr访问得到的结果都是一样的。

那么既然多个shred\_ptr之昂同一个实例对象会增加引用计数，如果是删除一个shred\_ptr呢？

#### 把shared\_ptr赋值为nullptr

如下是把已经存在的指针sa\_newptr手动赋值为nullptr，然后在通过另一个指针as\_new-tr2尝试调用对象的方法，因为前后的shared\_ptr都是指向的同一个对象，所以只要对象的引用计数不为0，那么实例对象仍然是可以访问的：

  cout << "-----------shared\_ptr指针赋空值，直到计数为0 strat------------\n";

  sa\_newptr = nullptr;

  sa\_newptr2->info();

  cout << "sa\_newptr2.use\_count(): " << sa\_newptr2.use\_count() << endl;

  sa\_newptr2 = nullptr;

  cout << "shared\_ptr引用计数降到0后，自动调用类型的析构函数.\n";

  cout << "sa\_newptr2.use\_count(): " << sa\_newptr2.use\_count() << endl;

  cout << "-----------shared\_ptr指针赋空值，直到计数为0 end------------\n";

输出如下：

-----------shared\_ptr指针赋空值，直到计数为0 strat------------

info信息,num=100

sa\_newptr2.use\_count(): 1

A的析构函数,num=100

shared\_ptr引用计数降到0后，自动调用类型的析构函数.

sa\_newptr2.use\_count(): 0

-----------shared\_ptr指针赋空值，直到计数为0 end------------

而通过输出也可以看出来，在给一个sa\_newptr赋值为空之后，用另外一个指针检查对异性的引用计数已经变为1了，也就是说当前对象只有sa\_newptr2一个指针拥有对象的指向了，然后再次把sa\_newptr2也赋值为空。再次观察sa\_newptr2的应用计数，发现计数已经是0了，并且随后自动调用了类型的析构函数销毁了对象。

所以，如果是多个shared\_ptr同时指向同一个对象，那么shared\_ptr的应用计数和指针的个数是一致的，实例对象有几个shred\_ptr指针，应用计数就是几。当一个个的把shared\_ptr手动赋值为nullptr后，相应的引用计数会减小，直到应用计数为0时，会自动调用类型的析构函数销毁对象。

### shared\_ptr的传递

shared\_ptr同样也存在着值传递和引用传递两种方式。

#### shrared\_ptr的指针值传递

有如下函数func，形参是shared\_ptr的指针值传递类型：

/\* shared\_ptr的指针值传递 \*/

void func(shared\_ptr<A> a) {

  cout << "进入值传递的函数，引用计数：" << a.use\_count() << endl;

  a->info();

  a->setNum(30); // 修改源数据的内容

}

调用如下：

  cout << "-------------shared\_ptr 的指针值传递 start ----------------\n";

  shared\_ptr<A> a\_trans{new A(5)};

  cout << "指针值传递后，a\_trans.info:";

  a\_trans->info();

  cout << "指针值传递前，a\_trans.use\_count:" << a\_trans.use\_count() << endl;

  func(a\_trans);

  cout << "指针值传递后，a\_trans.use\_count:" << a\_trans.use\_count() << endl;

  cout << "指针值传递后，a\_trans.info:";

  a\_trans->info();

  cout << "-------------shared\_ptr 的指针值传递 end ----------------\n";

输出：

-------------shared\_ptr 的指针值传递 start ----------------

A的带参构造函数,num=5

指针值传递后，a\_trans.info:info信息,num=5

指针值传递前，a\_trans.use\_count:1

进入值传递的函数，引用计数：2

info信息,num=5

指针值传递后，a\_trans.use\_count:1

指针值传递后，a\_trans.info:info信息,num=35

-------------shared\_ptr 的指针值传递 end ----------------

在这个例子中，现在主调函数中定义了一个shared\_ptr指针a\_trans指向new出来的A实例对象num=5，然后调用函数func并且把a\_trans作为参数传递，在进入func函数后检查应用计数发现增加了1。

然后func调用结束，返回主调函数，再次检查引用计数发现计数又闭函数中的引用计数减小了1。也就是说在值传递过程中，进入调用函数后会引发shared\_ptr增加1，在完成函数调用后，返回主调函数中时，应用计数又自动减小1，恢复到调用前的状态(在调用函数func中销毁对象的情况除外)。并且调用前后整个过程，指向保持不变始终是同一份对象，因为在调用前，调用中，以及调用结束后，访问对象的info方法得到的信息都是一样的。

#### shared\_ptr的引用传递

如下函数func2，参数时使用的shared\_ptr的引用：

/\* shared\_ptr的引用传递 \*/

void func2(shared\_ptr<A> &a) {

  cout << "进入引用传递的函数，引用计数：" << a.use\_count() << endl;

  a->info();

  a->setNum(40);

}

主调代码如下：

  cout << "-------------shared\_ptr 的指针引用传递 start ----------------\n";

  shared\_ptr<A> a\_ref = make\_shared<A>(60);

  cout << "指针引用传递前，a\_ref.info:";

  a\_ref->info();

  cout << "指针引用传递前，a\_ref.use\_count:" << a\_ref.use\_count() << endl;

  func2(a\_ref);

  cout << "指针引用传递后，a\_ref.use\_count:" << a\_ref.use\_count() << endl;

  cout << "指针引用传递后，a\_ref.info:";

  a\_ref->info();

  cout << "-------------shared\_ptr 的指针引用传递 end ----------------\n";

输出：

-------------shared\_ptr 的指针引用传递 start ----------------

A的带参构造函数,num=60

指针引用传递前，a\_ref.info:info信息,num=60

指针引用传递前，a\_ref.use\_count:1

进入引用传递的函数，引用计数：1

info信息,num=60

指针引用传递后，a\_ref.use\_count:1

指针引用传递后，a\_ref.info:info信息,num=100

-------------shared\_ptr 的指针引用传递 end ----------------

这里重点观察在调用func2之前，和进入func2函数中，以及返回主调函数后，期间shared\_ptr的引用计数都没有发生变化，也就是说，按照引用传递不会增加shared\_ptr的计数。

#### shared\_ptr调用类型的析构函数的时机

前面已经验证过在对象的shared\_ptr指针的应用计数减小到0过后，会自动调用对象的析构函数。如果是直接离开作用域会什么情况，是想unique\_ptr一样自动调用析构函数吗？

如下定义一个内部代码块{}，然后在内部代码块中创建shared\_ptr的指针will\_del指向new出来的A实例:

  {

    cout << "\*\*\*\*\*\*代码块内部的shared\_ptr strat\*\*\*\*\*\*";

    shared\_ptr<A> will\_del{new A(1)};

    shared\_ptr<A> another\_will\_del = will\_del;

    cout << "another\_will\_del.use\_count:" << another\_will\_del.use\_count()

         << endl;

  }

  cout << "\*\*\*\*\*\*代码块内部的shared\_ptr end\*\*\*\*\*\*";

输出：

\*\*\*\*\*\*代码块内部的shared\_ptr strat\*\*\*\*\*\*A的带参构造函数,num=1

another\_will\_del.use\_count:2

A的析构函数,num=1

\*\*\*\*\*\*代码块内部的shared\_ptr end\*\*\*\*\*\*A的析构函数,num=100

观察输出可以看出，在代码快中new出的A实例num=1，虽然有两个shared\_ptr指向了他，而且在离开代码块之前，引用计数是2，并没有减小到0，但是代码块结束后，仍然是调用了A实例num=1的析构函数。所以有如下结论：

* 一个实例对象的shared\_ptr个数，如果引用计数为0时，会自动调用析构函数，而使得计数为0的手段，可以手动不断的把对象的shared\_ptr指针赋值为空，直到所有的shared\_ptr都断开；
* 在离开shared\_ptr声明的作用域时，不管其应用计数是否为0，都会把shared\_ptr背后的对象自动销毁，会自动调用对象的析构函数。

#### 从函数中返回局部创建的shared\_ptr

如下函数get\_shared，在函数中创建shared\_ptr并指向new出的A类实例，然后把创建的额shared\_ptr使用return返回：

shared\_ptr<A> get\_shared() {

  shared\_ptr<A> a\_in\_block = make\_shared<A>(30);

  cout << "返回函数中创建的shared\_ptr前，a\_in\_block.use\_count:"

       << a\_in\_block.use\_count() << endl;

  return a\_in\_block;

}

主调函数如下：

int main(int argc, char const \*argv[]) {

  shared\_ptr<A> a\_in\_main = get\_shared();

  cout << "回到函数中创建的shared\_ptr前，a\_in\_main.use\_count:"

       << a\_in\_main.use\_count() << endl;

  /\* 在被调用的函数中创建的shared\_ptr，返回主调后，引用计数不会增加 \*/

  get\_shared()->info(); // shared\_ptr的链式访问

  return 0;

}

在主调函数中，使用一个相同类型的shared\_ptr接受函数返回的shared\_ptr。

输出如下：

A的带参构造函数,num=30

返回函数中创建的shared\_ptr前，a\_in\_block.use\_count:1

返回函数中创建的shared\_ptr前，a\_in\_main.use\_count:1

A的带参构造函数,num=30

返回函数中创建的shared\_ptr前，a\_in\_block.use\_count:1

info信息,num=30

A的析构函数,num=30

A的析构函数,num=30

这里重点观察在函数中创建的shared\_ptr的引用计数，再返回到主函数用a\_in\_main接收后，其计数值并没有变化。也就是说，在函数中创建的shared\_ptr，返回给主调函数，整个过程中引用计数一直保持不变。

而且不影响shared\_ptr会自动调用析构函数的特点。

## weak\_ptr的使用

和unique\_ptr和shared\_ptr相比，weak\_ptr重点在于处理循环引用。

### weak\_ptr处理循环引用

如下定义的类A:

// A class strat

class A {

public:

  explicit A(int n);

  A();

  A(const A &a);

  ~A();

  void setNum(int v);

  void setFriend(shared\_ptr<A> f);

  void setFriendWeak(shared\_ptr<A> f);

  void info() const; // 后置const限定函数内不能修改成员值

private:

  int num = 0;

  shared\_ptr<A> A\_frined; // 危险！这里使用shared\_ptr可能会存在循环依赖问题

  weak\_ptr<A> A\_friend\_weak; // 设置为weak\_ptr类型更安全

};

A::A(int n) : num(n) {

  std::cout << "A的带参构造函数,num=" << num << std::endl;

}

A::A() { std::cout << "A的默认构造函数，num=" << num << std::endl; }

A::A(const A &a) { "A的拷贝构造函数\n"; }

A::~A() { std::cout << "A的析构函数,num=" << num << std::endl; }

void A::info() const { std::cout << "info信息,num=" << num << std::endl; }

void A::setNum(int v) { num += v; }

void A::setFriend(shared\_ptr<A> f) { A\_frined = f; }

void A::setFriendWeak(shared\_ptr<A> f) { A\_friend\_weak = f; }

// 虽然成员A\_friend\_weak是weak\_ptr,不是shared\_ptr类型，但是可以自动从shared\_ptr转换为weak\_ptr

// 所以入参类型直接设置为shared\_ptr即可

// A class end

在这个自定义的类A中，注意到成员变量有一个类型为shared\_pt<A>的智能指针A\_fread和一个weak\_ptr<A>类型的智能指针A\_weak\_friend。

在面对循环引用时，sahred\_ptr的处理方式就会导致异常，如下主调函数：

{

  cout << "----------类成员数据的循环依赖 strat---------\n";

  shared\_ptr<A> ra1 = make\_shared<A>(11);

  shared\_ptr<A> ra2 = make\_shared<A>(22);

  // 危险！ 设置了循环依赖

  ra1->setFriend(ra2);

  // ra2->setFriend(ra1);

  // ra1和ra2之间的A\_friend成员存在循环依赖，会导致运行时ra1,ra2无法正常析构,内存泄漏

}

  cout << "----------类成员数据的循环依赖 end  ---------\n";

  cout << "------------------ main end ------------------\n";

在这个调用中，内部代码快中创建两个shared\_ptr<A>类型的指针ra1和ra2,然后调用了自身的set\_friend方法互相把对方设置为自己的数据成员。这样在ra1析构时会先析构成员ra2，但是析构ra2时，ra2中又有ra1成员，于是又会去转向ra1的析构…，如此循环往复下去…

输出：

----------类成员数据的循环依赖 strat---------

A的带参构造函数,num=11

A的带参构造函数,num=22

----------类成员数据的循环依赖 end ---------

------------------ main end ------------------

从输出可以看出来，只调用了创建对象时候的构造函数，但是离开代码块作用域后，ra1和ra2的析构函数并没有正常调用，造成内存泄漏。

然后如果是设置weak\_ptr类型的数据成员，如果是循环应用就不存在，如下：

{

    cout << "----------类成员数据的循环依赖 strat---------\n";

    shared\_ptr<A> ra1 = make\_shared<A>(11);

    shared\_ptr<A> ra2 = make\_shared<A>(22);

    // 危险！ 设置了循环依赖

    // ra1->setFriend(ra2);

    // ra2->setFriend(ra1);

    // ra1和ra2之间的A\_friend成员存在循环依赖，会导致运行时ra1,ra2无法正常析构,内存泄漏

    // OK! 成员A\_fiend\_weak是weak\_ptr类型，可以循环依赖

    ra1->setFriendWeak(ra2);

    ra2->setFriendWeak(ra1);

    // 设置weak\_ptr成员，如果有循环依赖，也可以正常析构

  }

  cout << "----------类成员数据的循环依赖 end  ---------\n";

ra1和ra2不互相设置shared\_ptr<A>类型的成员为自己的组合成员变量，而是把对方转换为weal\_prt<A>类型的然后在设置为自己的成员变量，这里注意一点，虽然是调用的set\_friendWeak方法，形参都是shared\_ptr类型，但是如果在赋值的时候，实际的目标对象是weak\_ptr类型，shared\_ptr会自动转换类型过去的，所以无须担心形参的类型问题。

输出：

----------类成员数据的循环依赖 strat---------

A的带参构造函数,num=11

A的带参构造函数,num=22

A的析构函数,num=22

A的析构函数,num=11

----------类成员数据的循环依赖 end ---------

可以看到，如果是weal\_ptr类型的成员，ra1和ra2互相循环嵌套，在析构时也可以正常析构的。

所有，如果在使用过程中，存在着类成员循环引用的问题，最好的办法是把成员类型设置为weak\_ptr。

### weak\_ptr的使用

#### weak\_ptr的使用方式1——从shared\_ptr初始化创建

weak\_ptr无法直接定义，直接存在，需要依赖于shared\_ptr:

  shared\_ptr<A> sa1 = make\_shared<A>(10);

  weak\_ptr<A> weaka(sa1); // weak\_ptr依赖shared\_ptr

  cout << "sa1.use\_count:" << sa1.use\_count() << endl;

  cout << "weaka.use\_count:" << weaka.use\_count() << endl;

  /\* shared\_ptr转成weak\_ptr不影响引用计数 \*/

定义时使用weak\_ptr去包装已有的shared\_ptr智能指针，输出如下：

A的带参构造函数,num=10

sa1.use\_count:1

weaka.use\_count:1

同时从输出可以看出，使用weak\_ptr包装shared\_ptr之后，引用计数并不变化。

#### weak\_ptr的使用方式2——从shared\_ptr调用.lock()抓换位weak\_ptr

如下，通过对已有的shared\_ptr调用lock()方法，把shared\_ptr转换为weak\_ptr:

  /\* weak\_ptr调用.lock()转换为shared\_ptr \*/

  shared\_ptr<A> sa\_locked = weaka.lock();

  cout << "-----调用.lock()转换为shared\_ptr智能指针后-----\n";

  cout << "这里sa1/weaka/sa\_locked其实包装的是同一份内存\n";

  cout << "sa1.use\_count:" << sa1.use\_count() << endl;

  cout << "weaka.use\_count:" << weaka.use\_count() << endl;

  cout << "sa\_locked.use\_count:" << sa\_locked.use\_count() << endl;

  /\* weak\_ptr转换成shared\_ptr会增加引用计数+1 \*/

输出：

-----调用.lock()转换为shared\_ptr智能指针后-----

这里sa1/weaka/sa\_locked其实包装的是同一份内存

sa1.use\_count:2

weaka.use\_count:2

sa\_locked.use\_count:2

从输出可以看出，shared\_ptr调用lock()后，转换为weak\_ptr，同时引用计数会相应的增加1.