
第五课 类和对象（运算符重载）

内容概述

1. 运算符重载规则
2. 输入输出运算符重载
3. 一元运算符重载
4. 二元运算符重载
5. 关系运算符重载
6. 类型转换运算符
7. 函数调用运算符
8. 函数对象
9. lambda表达式
10. function、bind
11. 智能指针
12. shared_ptr
13. weak_ptr
14. unique_ptr
15. operator new/delete
16. allocator
17. 小练习

运算符重载: 引入

当运算符作用于类类型的运算对象时, 可以通过**运算符重载**重新定义该运算符的含义。
明智地使用运算符重载令程序更易于编写和阅读。

```
class Point {
public:
    Point(int _x = 0, int _y = 0)
        :x(_x), y(_y) {}
private:
    int x, y;
};
int main() {
    int i1 = 10, i2 = 20;
    int i3 = i1 + i2; //加法运算
    string s1 = "abc", s2 = "123";
    string s3 = s1 + s2; //string类实现的 运算符重载

    Point p1(10, 20), p2(20, 30);
    //Point p3 = p1 + p2; //不行? 如何做?
    //Point p3 = point_add(p1, p2); //不如上面的直观
    //Point p3 = p1.point_add(p2); //不如上面的直观
    return 0;
}
```

运算符重载的两种形式:

a + b ==> 1: operator+(a, b); //普通函数

a + b ==> 2: a.operator+(b); //成员函数

```
Point operator+(const Point &a, const Point &b) {
    int x1 = a.get_x() + b.get_x();
    int y1 = b.get_y() + b.get_y();
    Point tmp(x1, y1);
    return tmp;
} //普通函数
```

```
Point Point::operator+(const Point &b) {
    int x1 = x + b.x;
    int y1 = y + b.y;
    Point tmp(x1, y1);
    return tmp;
} //Point类的成员函数
```

运算符重载: 规则1

一: C++中大多数运算符是可以重载的:

算术运算符: + - * / % ++ --

位操作运算符: & | ~ ^ << >>

逻辑运算符: ! && ||

比较运算符: < > >= <= == !=

赋值运算符: = += -= *= /= %= &= |= ^= <<= >>=

其他运算符: [] () -> , new delete new[] delete[] ->*

不能重载的运算符有 . 和 .* 和 ?: 和 :: 和 sizeof

(成员运算符、成员对象选择符、条件运算符、域解析运算符、类型大小运算符)

二: 不能自创运算符, 不要违背运算符本身的含义:

比如python里面的**是指数运算, c++里面没有**运算符, 所以不能自创。
原来是+就是+, 运算符重载本来就是为了易于编写和阅读, 不能起反作用

三: 重载不能改变运算符的优先级和结合性:

比如: * / 优先级比 + - 高, $x = y + z * a$; 肯定是 $x = (y + (z * a))$;

比如: = 是右结合。 $a = b = c$; 肯定是 $a = (b = c)$;

运算符重载: 规则2

四: 运算对象的个数不能改变, 运算符重载不能有默认参数 (除了()运算符外):

比如: *运算符, 两个运算对象就是乘法, 一个运算对象就是解引用。

假如有默认参数? 二义性。例:

```
Point operator*(const Point &a, const Point &b); //乘法
Point &operator*(const Point &a); //解引用
```

五: 至少有一个操作数是自定义类。

假如都是内置类型, 比如: `int i, j; i + j` 这样的运算是不能重载的。

`string ss = "abc" + "123";` 错误。

六: 某些运算符不应该被重载:

运算符重载, 本质上是一次函数调用。

`&&` || 重载的话, 无法保留内置运算符的短路功能, 为了避免引起误用, 不建议重载。

, (逗号运算符) `&` (取地址) 一般也不重载。

运算符重载时, 要注意返回类型。

如: `operator+` 以对象值类型返回, `operator=` 以引用返回。

输入输出运算符重载

期望 `cin >> Point` 类对象，`cout << Point` 类对象 这样直观的方式来输入输出类对象。

因为 `cin cout` 类，是系统给出的，无法再去给它增加成员函数，所以只能使用普通函数（非成员函数）重载的输入输出函数，一般都声明为该类的友元函数。（方便访问类对象的成员变量）

```
class Point { //申明友元
    friend istream &operator >> (istream& in, Point& a);
    friend ostream &operator<<(ostream& out, const Point& a);
public:
    Point(int _x = 0, int _y = 0)
        :x(_x), y(_y) {}
    void print() { //原来的输出函数
        cout << "(" << x << ", " << y << ")" << endl;
    }
private:
    int x, y;
};
```

```
istream &operator >> (istream& in, Point& a) {
    cout << "input Point x y:" << endl;
    in >> a.x >> a.y;
    return in; //输入重载
}
```

```
ostream &operator<<(ostream& out, const Point& a) {
    cout << "(" << a.x << ", " << a.y << ")" << endl;
    return out; //输出重载
}
```

```
int main() {
    Point p1,p2(100,200);
    cin >> p1;
    p1.print();
    cout << p1 << endl; //和上面效果一样
    cout << p1 << p2 << endl; //还可以这样
    return 0;
}
```

```
input Point x y:
10 20
<10,20>
<10,20>
<10,20> <100,200>
请按任意键继续.
```

一元运算符 - (负号)

负号和减号：只有一个运算对象时是负号，有两个运算对象时是减号。负号运算得到一个**临时量**。

```
#include <iostream>
using namespace std;
class Point {
    ② friend Point operator-(const Point& a);
public:
    Point(int _x = 0, int _y = 0)
        :x(_x), y(_y) {}
    ① /*const*/ Point operator-()const;
private:
    int x, y;
};
① /*const*/ Point Point::operator-()const {
    return Point(-x, -y);
}
② Point operator-(const Point& a) {
    return Point(-a.x, -a.y);
}
```

```
int main() {
    Point p1(10,20);
    cout << p1 << endl;
    cout << -p1 << endl;
    //假如返回的是 const, 则必须用const引用接收
    /*const*/ Point &p2 = -p1;
    Point p3 = -p1; //值拷贝就无所谓
    cout << p2;
    return 0;
}
```

```
<10,20>
<-10,-20>
<-10,-20>
```

1: 成员函数 负号

2: 普通函数 负号

1和2 二义性冲突 (所以1, 2只能选1个)

函数返回值是否 const 的含义是什么?

二元运算符 + (加号)

期望:

Point类对象 + Point类对象;

Point类对象 + int;

int + Point类对象;

连加运算: int + Point + Point...

```
#include <iostream>
using namespace std;
class Point {
    .....
    friend Point operator+(const Point& a, const Point& b);
public:
    Point(int _x = 0, int _y = 0)
        :x(_x), y(_y) {}
    //Point operator+(const Point &a) const;
private:
    int x, y;
};
```

```
//Point Point::operator+(const Point &a) const {
//    return Point(x + a.x, y + a.y);
//}
```

一元运算符习惯上写为成员函数

二元运算符习惯上写为非成员函数

```
Point operator+(const Point& a, const Point& b) {
    return Point(a.x + b.x, a.y + b.y);
}
Point operator+(const Point& a, int i) {
    return a + Point(i, 0);
}
Point operator+(int i, const Point& a) {
    return a + Point(i, 0);
}
```

```
int main() {
    Point p1(1,2), p2(3,4);
    cout << p1 + p2 << endl; //(4,6)
    cout << 10 + p1 + p2 << endl; //(14,6)
    return 0;
}
```

<4,6>
<14,6>
请按任

复合赋值运算符+=

赋值运算符：= 复合赋值运算符： += -= *= /= %= &= |= ^= <<= >>=

赋值运算符=只能是成员函数。

复合赋值运算符属于二元运算符，习惯上将它定义为成员函数。

```
#include <iostream>
using namespace std;
class Point {
    .....
public:
    Point(int _x = 0, int _y = 0)
        :x(_x), y(_y) {}
    Point &operator+=(const Point &a);
private:
    int x, y;
};
Point & Point::operator+=(const Point &a) {
    x += a.x;
    y += a.y;
    return *this;
}
```

此处返回值和返回引用的区别？

```
int main() {
    int a = 2, b=3;
    a += b += 2;
    // a += (b += 2)
    //结果： b = 5, a = 7
    cout << b << a << endl; // 5 7
    (a += b) = 10; // a += b 返回 a
    cout << a << endl; //10
    Point p1(1, 2), p2(3, 4), p3(5,9);
    p1 += p2;
    cout << p1 << endl; // (4,6)
    (p1 += p2) = p3; // (p1 += p2) 返回的是 p1的引用
    cout << p1 << endl; // (5,9)
    p1 += p2 = p3; //右结合律, p1 += (p2 = p3)
    cout << p1 << endl; // (10,18)
    return 0; }
```

前++和后++

A& operator++(); //前置++ A operator++(int); //后置++ 注意：后置++会产生临时量。
显式调用前置++: a.operator++(), 显式调用后置++: a.operator++(0)

```
class Point {
    .....
    //friend Point &operator++(Point &a);
    //friend Point operator++(Point &a, int);
public:
    .....
    Point &operator++();
    Point operator++(int);
private:
    int x, y;
};
Point & Point::operator++() {
    x += 1;           前++
    y += 1;
    return *this;
}
/*const*/Point Point::operator++(int) {
    Point tmp = *this;   后++
    x += 1;
    y += 1; //上面2句也可写为: ++*this;
    return tmp;
}

//Point &operator++(Point &a) {
//    a.x += 1;
//    a.y += 1;
//    return a;
//}
//Point operator++(Point &a, int) {
//    Point tmp = a;
//    a.x += 1;
//    a.y += 1;
//    return tmp;
//}

int main() {
    Point p1(1, 2);
    cout << p1++ << endl; //(1,2)
    cout << ++p1 << endl; //(3,4)
    return 0;
}
(1,2)
(3,4)
请按任
```

下标运算符 []

一般都会定义非常量版本和常量版本。下标运算符重载必须是成员函数。

当作用于一个常量对象时，下标运算符返回常量引用确保不会给返回的对象赋值。

```
#include <iostream>
#include <cstring>
using namespace std;
class myString {
    friend ostream &operator<<(ostream& out, const myString& a);
public:
    myString(const char* _ps = NULL) {
        .....
    }
    .....
    char &operator[](int i) { return ps[i]; }
    const char &operator[](int i) const { return ps[i]; }
private:
    char *ps;
};
```

```
ostream &operator<<(ostream& out,
                    const myString& a) {
    out << a.ps;
    return out;
}

int main() {
    myString s1("abc");
    s1[1] = 'B';
    cout << s1 << endl; //aBc
    const myString s2("abc");
    cout << s2[1] << endl; //b
    return 0;
}
```

aBc
b
请按

关系运算符== != < >

```
#include <iostream>
#include <string>
using namespace std;
class Person {
public:
    Person(int _age, const string& _name)
        :age(_age), name(_name){}
    bool operator==(const Person& rhs) const;
    bool operator!=(const Person& rhs) const;
    bool operator<(const Person& rhs) const;
private:
    int age;
    string name;
};
bool Person::operator==(const Person& rhs) const {
    return age == rhs.age && name == rhs.name;
}
bool Person::operator!=(const Person& rhs) const {
    return !(*this == rhs);
}
```

```
bool Person::operator<(const Person& rhs) const {
    if (age < rhs.age)
        return true;
    if (age == rhs.age && name < rhs.name)
        return true;
    return false;
}
int main() {
    Person p1(20, "alex");
    Person p2 = p1;
    Person p3(20, "bob");
    cout << (p1 == p2) << endl; //1
    cout << (p1 != p3) << endl; //1
    cout << (p1 < p2) << endl; //0
    cout << (p1 < p3) << endl; //1
    return 0;
}
```

1
1
0
1
请接

类型转换运算符

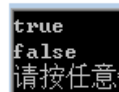
类型转换运算符是类的一种**特殊成员函数**，它负责将一个类类型转换成其他类型。

表示形式：**operator type() const**；如：**operator int()**；转为int类型；**operator bool()**；转为bool类型。

注意：类型转换的方向，**explicit**关键字的作用。

```
#include <iostream>
#include <vector>
using namespace std;
class B {};
class A{
public:
    A(int i=0):data(i){ } //int -> A
    explicit A(const B&) { /*...*/ } //B --> A
    operator int()const { return data; } //A -->int
    explicit operator B()const{return B();} //A --> B
    operator bool()const { return data; } //A --> bool
private:
    int data;
};

//转换为bool类型
if ( A(1) ) cout << "true\n";
if ( !A() ) cout << "false\n";
```



```
//1.1 没有explicit修饰的构造函数,参数类型 ==> 本类类型
A a1(1); // int -> A
A a2 = 2; // int -> A
```

```
//1.2 有explicit修饰的构造函数,参数类型 ==> 本类类型
B b;
A a3(b); // B -> A
//A a4 = b; //因为 explicit,隐式转换不行
A a5 = static_cast<A>(b); //显式转换ok
```

```
//2.1 没有explicit修饰的类型转换运算符, 本类类型 ==> 其他类型
A a;
int i1(a); // A -> int
int i2 = a; // A -> int
```

```
//2.2 有explicit修饰的类型转换运算符, 本类类型 ==> 其他类型
A aa;
B b1(aa); // A -> B
//B b2 = aa; // 因为explicit,隐式转换不行
B b3 = static_cast<B>(aa); //显示转换ok
```

类型转换二义性

当类中同时定义了类型转换运算符和重载运算符时，要特别小心二义性错误。

```
class Point {
    friend ostream& operator<<(ostream&, const Point&);
public:
    explicit Point(int a = 0, int b = 0) :x(a), y(b) {}
    Point operator+(const Point& rhs) {
        int tmp_x = x + rhs.x;
        int tmp_y = y + rhs.y;
        return Point(tmp_x, tmp_y);
    }
    /*explicit*/ operator int()const {return x;}
private:
    int x;
    int y;
};
ostream& operator<<(ostream& out, const Point& p) {
    out << "(" << p.x << ", " << p.y << ")\n";
    return out;
}
```

```
int main() {
    Point p1(1, 2);
    cout << (p1 + 3) << endl;
    return 0;
}
```

```
explicit Point(int, int);      4 请任
operator+(int, int);
```

```
Point(int, int);             <4,2>
explicit operator int();     请接任 Point.operator+(const Point&);
```

都加上 `explicit`, 因为没有 `operator+(Point, int)` 函数, 不行
都不加 `explicit`, 两种转换方式都可以, **二义性错误**, 不行

函数调用运算符

有一个vector<string>,要统计其中字符串长度小于3的元素个数。

```
#include <iostream>
#include <cstring>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

class Func {
public:
    Func(int len = 3) :length(len) {}
    bool operator()(const string& s) {
        if (s.size() < length) return true;
        return false;
    }
private:
    int length;
};

bool less3(const string& s) {
    if (s.size() < 3) return true;
    return false;
}
```

```
int main() {
    vector<string> vs = { "abc","as","a","aff" };
    int res = count_if(vs.begin(), vs.end(), less3);

    cout << res << endl; //输出 2

    //问题：假如要求输出长度小于4的元素个数怎么办？
    //方案1：修改less3函数，里面的 3修改为4（不方便，太麻烦）
    //方案2：less3函数增加一个参数，但是count_if函数中的第3个参数是函数指针，
    //          只允许有一个参数，（不行）
    //方案3：搞一个全局变量，less3函数中的 3用全局变量替代（不方便,容易出错）

    //仿函数：通过重载 operator() 函数调用运算符
    res = count_if(vs.begin(), vs.end(), Func()); //默认构造是3,注意这里的()
    cout << res << endl; //输出2
    res = count_if(vs.begin(), vs.end(), Func(4));
    cout << res << endl; //输出4
    return 0;
}
```

函数对象

(函数调用运算符必须是成员函数)

返回类型

operator()(参数);

类定义了函数调用运算符(), 则该类的对象称作函数对象 (function object)

仿函数:
行为像函数一样

求绝对值

```
struct absInt {
    int operator()(int val) const {
        return val > 0 ? val : -val;
    }
};
int main() {
    int a = -2;
    absInt abs;
    cout << abs(a) << endl; // 输出2
    cout << absInt()(-3) << endl; //输出3
    return 0;
}
```

```
#include <iostream>
#include <cstring>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
class PrintString {
public:
    PrintString(char c = ' ') : sep(c) {}
    void operator()(const string& s) const {
        cout << s << sep;
    }
private:
    char sep;
};
int main() {
    vector<string> vs = { "abc", "as", "a", "aff" };
    //for_each 函数的第三个参数 也是一个函数指针, 只允许有一个参数
    for_each(vs.begin(), vs.end(), PrintString()); //默认分隔符
    cout << endl;
    for_each(vs.begin(), vs.end(), PrintString(',')); //分隔符是,
    cout << endl;
    PrintString fps1 = PrintString('|');
    for_each(vs.begin(), vs.end(), fps1); //分隔符是|
    cout << endl;
    return 0;
}
```

输出 vector<string> 中的元素, 要求用不同的分隔符

```
abc as a aff
abc,as,a,aff
abc|as|a|aff|
请按任意键继续
```


lambda表达式

lambda表达式，是一个函数对象(匿名的函数对象)。理解为：未命名的内联函数。(匿名函数)
格式：[捕获列表] (参数列表) -> 返回值类型 { 函数体 } (参数列表) 和 ->返回值类型 可以省略
auto f = [] {return 2;} cout << f() <<endl; //2

捕获列表：只用于局部非static变量，lambda可以直接使用局部static变量和它所在函数之外声明的名字。

```
[]          // 不捕获任何外部变量
[=]         // 以值的形式捕获所有外部变量
[&]         // 以引用形式捕获所有外部变量
[x, &y]     // x 以传值形式捕获，y 以引用形式捕获
[=, &z]     // z 以引用形式捕获，其余变量以传值形式捕获
[&, x]     // x 以值的形式捕获，其余变量以引用形式捕获
```

lambda中使用this指针(成员函数中使用lambda):

对于[=]或[&]的形式，lambda表达式可以直接使用 this 指针。但是，对于[]的形式，如果要使用 this 指针，必须显式传入：[this]() { this->someFunc(); }();

第一个例子，用lambda表达式:

```
res = count_if( vs.begin(), vs.end(), [](const string& s) ->bool {if (s.size() < 3) return true; return false; } );
```

lambda表达式

lambda是通过匿名的函数对象来实现的，因此可以把lambda看作是对函数对象在使用方式上进行的简化。当代码需要一个简单的函数，并且不会在其他地方使用，可以用lambda来实现，作用类似于匿名函数。如果需要多次调用，并且需要保存某些状态的话，则使用函数对象更好。

```
int g_i = 10;
int main() {
    static int s_i = 20;
    int i1 = 10;
    auto f1 = [] {
        cout << g_i << s_i << endl; //全局,static变量可以用
        //cout << i1 << endl; 用不了i1
    };
    f1(); //输出10 20
    auto f2 = [i1](int i) {
        cout << i1 << i << endl; //值传递
    };
    i1 = 100;
    f2(0); //输出 10 0, 此处注意: i1是定义lambda时的i1,值拷贝
    auto f3 = [&i1] {cout << i1 << endl; }; //引用传递
    i1 = 20;
    f3(); //输出20 引用传递
    int i2 = 30;
    [&] {cout << i1 << i2 << endl; }(); //直接全部引用局部变量
    //输出 20 30
    return 0;
}
```

```
1020
100
20
2030
请按任
```

```
class NoName {
public:
    NoName(int& sum, int num)
        :sum_(sum), num_(num) {}
    int operator()(int i)const {
        return sum_ += num_ + i;
    }
private:
    int& sum_;
    int num_;
};
int main() {
    int sum = 1, num = 2;
    NoName na = NoName(sum, num);
    cout << na(3) << endl; //6
    cout << sum << endl; //6
    sum = 1, num = 2;
    int tmp = [&sum, num](int i) {
        return sum += num + i; }(3);
    cout << tmp << endl; //6
    cout << sum << endl; //6
    return 0;    return 0;
}
```

```
6
6
6
6
请
```

标准库函数对象

```
#include <iostream>
#include <vector>
#include <string>
#include <functional>
#include <algorithm>
using namespace std;
class Person {
    friend bool cmp(const Person& lhs, const Person& rhs);
    friend class cmp1;
public:
    Person(int _age,const string& _name):age(_age),name(_name){}
    bool operator<(const Person& rhs)const {
        if (age < rhs.age) return true;
        if (age == rhs.age && name < rhs.name) return true;
    } return false;
    bool operator>(const Person& rhs)const {
        if (age > rhs.age) return true;
        if (age == rhs.age && name > rhs.name) return true;
    } return false;
    string to_str()const {
        return string("(" + to_string(age) + "," + name + ")");
    }
private:
    int age; string name;
};
void print(const vector<Person>& vec) {
    for_each(vec.begin(), vec.end(), [](const Person& pn){
        cout << pn.to_str() << "\t"; });
}
```

标准库函数对象：

less<Type>, greater<Type>, less_equal<Type>,
greater_equal<Type>, equal_to<Type>等等

```
bool cmp(const Person& lhs, const Person& rhs) {
    if (lhs.age < rhs.age) return true;
    if (lhs.age == rhs.age && lhs.name < rhs.name) return true;
} return false;
struct cmp1 {
    bool operator()(const Person& lhs, const Person& rhs) {
        if (lhs.age < rhs.age) return true;
        if (lhs.age == rhs.age && lhs.name < rhs.name) return true;
        return false;
    }
};
int main() {
    vector<Person> vec = { {20,"ZhangSan"},{21,"Lisi"},{20,"WangWu"} };
    sort(vec.begin(), vec.end()); //默认是调用 less<Person> 等价于下面这句
    print(vec); cout << endl;
    sort(vec.begin(), vec.end(), less<Person>()); //从小到大排序
    print(vec); cout << endl;
    sort(vec.begin(), vec.end(), greater<Person>()); //默认是调用 less<Person>
    print(vec); cout << endl;
    sort(vec.begin(), vec.end(), cmp); //比较函数是 cmp
    print(vec); cout << endl;
    sort(vec.begin(), vec.end(), cmp1()); //比较函数是 函数对象cmp1
    print(vec); cout << endl;
    sort(vec.begin(), vec.end(), [](const Person& lhs,const Person& rhs)->bool{
        if (lhs < rhs) return true;
        return false;
    }); //比较函数是 lambda函数对象
    print(vec); cout << endl;
    return 0;
}
```

```
<20,WangWu> <20,ZhangSan> <21,Lisi>
<20,WangWu> <20,ZhangSan> <21,Lisi>
<21,Lisi> <20,ZhangSan> <20,WangWu>
<20,WangWu> <20,ZhangSan> <21,Lisi>
<20,WangWu> <20,ZhangSan> <21,Lisi>
<20,WangWu> <20,ZhangSan> <21,Lisi>
请按任意键继续...
```

可调用对象和function

```
#include <iostream>
#include <vector>
#include <functional>
using namespace std;

int add(int a, int b) { return a + b; }

class sub {
public:
    int operator()(int a, int b) {
        return a - b;
    }
};
```

可调用的对象：函数，函数指针，重载了函数调用运算符的类，lambda表达式，bind创建的对象。

标准库function类型，是一个模板类，是各种可调用实体的一种类型安全的封装。

```
//function用法
int main() {
    function<int(int, int)> f1 = add;
    function<int(int, int)> f2 = sub();
    function<int(int, int)> f3 =
        [](int a, int b) {return a*b; };
    cout << sub()(3, 2) << endl; //1
    cout << f2(10, 1) << endl; //9
    cout << f3(2, 4) << endl; //8
    cout << "-----\n";
    vector<int(*)(int, int)> vec1;
    vec1.push_back(add);
    //vec1.push_back(sub()); //不行了
    vec1.push_back([](int a, int b) {return a * b; });

    vector<function<int(int, int)>> vec2;
    vec2.push_back(add);
    vec2.push_back(sub());
    vec2.push_back([](int a, int b) {return a * b; });
    cout << vec2[0](3, 5) << endl; // 3+5 =8
    cout << vec2[1](3, 5) << endl; // 3-5 =-2
    cout << vec2[2](3, 5) << endl; // 3*5 =15
    return 0;
}
```

bind函数

std::bind函数，通用的函数适配器，它接受一个可调用对象，生成一个新的可调用对象。

```
#include <iostream>
#include <vector>
#include <string>
#include <functional>
#include <algorithm>
using namespace std;

void f1(int a, int b, int c) {
    cout << "a=" << a;
    cout << " b=" << b;
    cout << " c=" << c << endl;;
}

class A {
public:
    int add(int a, int b) { return a + b; }
};

ostream& print(ostream &out, const string &s,
               char sep) {
    return out << s << sep;
}
```

```
int main() {
    auto fun = bind(f1, std::placeholders::_2, 100, std::placeholders::_1);
    //第1个参数是 接受的可调用对象
    //std::placeholders::_1 _2 _3是新可调用对象的参数位置
    fun(10, 20);

    function<void(int)> f = bind(f1, 100, std::placeholders::_1, 300);
    f(200);

    //类的成员函数也可使用bind重新生成一个新的可调用对象
    A a;
    a.add(10, 20);
    function<int(int, int)> fc = bind(&A::add, a, std::placeholders::_1,
                                     std::placeholders::_2);
    cout << fc(10, 20) << endl;

    //不可拷贝的参数用 引用ref()
    vector<string> ss = { "ABC", "12345", "China" };
    for_each(ss.begin(), ss.end(), bind(print, ref(cout),
                                         std::placeholders::_1, ','));
    return 0;
}
```

```
a=20 b=100 c=10
a=100 b=200 c=300
30
ABC,12345,China,请按任
```

智能指针：引入

RAII (Resource Acquisition Is Initialization), 也称为“**资源获取就是初始化**”, 是C++语言的一种管理资源、避免泄漏的惯用法。C++标准保证任何情况下, 已构造的对象最终会销毁, 即它的析构函数最终会被调用。简单的说, RAII 的做法是使用一个对象, 在其构造时获取资源, 在对象生命期控制对资源的访问使之始终保持有效, 最后在对象析构的时候释放资源。

```
void f1() {
    int *pi = new int(20);
    // do something...
    if (*pi > 0) {
        //do something...
        return; //1.忘记delete
    }
    delete pi;
}
```

//2.调用者不清楚要不要释放资源
//函数声明：不看实现或文档时：
//调用者不知道返回的这个指针要不要释放
const char * get_name();

```
//这种情况, 调用者不需要去管理资源
const char * get_name1() {
    static char name[100];
    //do something...
    return name;
}
```

```
//这样的情况, 调用者要管理资源
const char* get_name2() {
    char * name = new char[100];
    //do something...
    return name;
}
```

```
//返回指针, 意义不清楚
//通常这样的写法, 调用者不需要管理资源
//因为指针是通过参数传入的
char* get_name3(char *v, int length){
    //do something
    return v;
}
```

```
//3.出现异常时
void f3() {
    int * pi = new int(20);
    //do something...
    throw 1;
    delete pi;
}
```

思考：有什么方法能有效地解决上面的这些问题？

智能指针：简单模拟

智能指针是一个类，它使用了RAII技术，在该类的构造函数中传入一个普通指针，析构函数中释放传入的指针。

1. 定义一个类来封装资源的分配与释放，在构造函数中完成资源的分配和初始化，在析构函数中完成资源的清理，它保证了资源正确的初始化和释放。
2. 智能指针不是指针，它实际上是一个（模板）类，由智能指针实例化出来的对象具有和常规指针相似的行为，重点是智能指针负责自动的释放所指对象。
3. 由于智能指针的类对象都是栈上的对象，所以当函数（或程序）结束时会自动被释放。

```
class T {
public:
    T(int i = 0) :data(i) { cout << "T构造\n"; }
    ~T() { cout << "T析构\n"; }
private:
    int data;
};

class SmartPointer {
public:
    SmartPointer(T* p = nullptr) :ptr(p) { }
    ~SmartPointer() { delete ptr; }
private:
    T * ptr;
};
```

```
int main() {
    T * pt1 = new T(1);
    SmartPointer spt1(pt1);
    T * pt2 = new T(2);
    SmartPointer spt2 = pt2;
    //函数结束时, pt1,pt2管理的两个 T 对象都会析构
    cout << "-----\n";
    return 0;
}
```

T构造
T构造
T析构
T析构
请按任

使用SmartPointer栈对象来管理堆空间中T的资源。
保证了堆空间中T资源的释放。

智能指针：*和->重载，行为像指针

```
class T {
public:
    T(int i = 0) :data(i) { cout << "T构造\n"; }
    ~T() { cout << "T析构\n"; }
    void show() const { cout << data << endl; }
private:
    int data;
};

class SmartPointer {
public:
    SmartPointer(T* p = nullptr) :ptr(p) { }
    ~SmartPointer() { delete ptr; }
    T& operator*() {
        assert(ptr);
        return *ptr;
    }
    T* operator->() {
        assert(ptr);
        return ptr;
    }
    operator bool()const {return ptr != nullptr;}
private:
    T * ptr;
};
```

```
int main() {
    T* p1 = new T(2);
    (*p1).show();
    p1->show();
    if (p1) cout << "p1 ok\n";
    delete p1;
    SmartPointer pd(new T(1));
    (*pd).show(); // *解引用操作
    pd->show(); // ->操作
    if (pd) cout << "pd ok\n";
    return 0;
}
```

```
T构造
2
2
p1 ok
T析构
T构造
1
1
1
pd ok
T析构
请按任
```

具有和常规指针相似的行为：**重载 *和->运算符**。

智能指针：引用计数

引用计数：

多个智能指针指向同一个对象。

上页的写法，无法解决这样的事情：

```
T* p1 = new T(2); T* p2 = p1; //多个指针指向一个对象
```

```
SmartPointer pd1(new T(1)); SmartPointer pd2 = pd1;
```

系统自动生成的拷贝构造，浅拷贝，会出现重析构问题。

解决方法：**引用计数**

```
#include <iostream>
#include <cassert>
using namespace std;
class T {
public:
    T(int i = 0) :data(i) { cout << "T构造\n"; }
    ~T() { cout << "T析构\n"; }
    void show()const { cout << data << endl; }
private:
    int data;
};

int main() {
    SmartPointer pd1(new T(1));
    SmartPointer pd2 = pd1; //拷贝构造
    SmartPointer pd3;
    pd3 = pd2; //赋值
    //退出时，只析构了1次
    return 0;
}
```

T构造
T析构
请按任

```
class SmartPointer {
public:
    SmartPointer(T* p = nullptr)
        :ptr(p), count(new int(ptr ? 1 : 0)) { }
    ~SmartPointer() {
        if (--*count) <= 0) {
            delete ptr; delete count;
        }
    }
    SmartPointer(const SmartPointer& other)//拷贝构造
        :ptr(other.ptr), count(&(++*other.count)) { }
    SmartPointer& operator=(const SmartPointer& other) {
        if (this == &other) return *this; //赋值操作
        ++*other.count;
        if (--*this->count <= 0) {
            delete ptr;
            delete count;
        }
        ptr = other.ptr;
        count = other.count;
        return *this;
    }
    T& operator*() { assert(ptr); return *ptr;}
    T* operator->() { assert(ptr); return ptr;}
    operator bool()const { return ptr != nullptr; }
private:
    T * ptr;
    int * count;
};
```

智能指针: shared_ptr

```
#include <iostream>
#include <memory>
using namespace std;
class T {
public:
    T(int i = 0) :data(i) { cout <<"T构造\n"; }
    ~T() { cout <<"T析构\n"; }
    void show()const { cout << data << endl; }
private:
    int data;
};
//值传递: 拷贝, 引用计数+1
void print1(shared_ptr<T> p) {
    cout <<"值: " << p.use_count() << endl;
    p->show();
}
//引用传递: 引用计数不会+1
void print2(const shared_ptr<T> &p) {
    cout <<"引用: " << p.use_count() << endl;
    p->show();
}
void print(const T& obj) {
    cout <<"对象的引用\n";
    obj.show();
}
void del_T_arr(T* parr) { //删除器
    delete[] parr;
}
```

int main() { **shared_ptr用法(接口)**

```
shared_ptr<T> p1; //空指针
//use_count() 引用计数(同时有几个指针指向对象)
cout << p1.use_count() << endl; //0
//unique(),use_count()==1返回true,否则false
cout << p1.unique() << endl; //0

//shared_ptr<T> p2= new T(1);//不行,explicit构造
shared_ptr<T> p2(new T(10));
cout << p2.use_count() << endl; //1
cout << p2.unique() << endl; //1
(*p2).show(); //10 行为像指针 *
p2->show(); //10 行为像指针 ->
if (p2) cout << "p2 not null!\n";

shared_ptr<T> p3 = p2; //拷贝构造
shared_ptr<T> p4(p2); //拷贝构造
cout << p2.use_count() << endl; //3
cout << p4.use_count() << endl; //3
cout << p3.unique() << endl; //0

//推荐用这样的方式初始化
shared_ptr<T> p5 = make_shared<T>(20);
//返回智能指针中保存的裸指针, 谨慎使用!
T* p = p5.get();
p->show(); //20
```

```
0
0
T构造
1
1
10
10
p2 not null!
3
3
0
T构造
20
2
2
T构造
1
30
20
值: 2
30
引用: 1
30
对象的引用
30
T构造
T构造
T构造
T构造
T构造
T构造
9
```

```
shared_ptr<T> p6;
p6 = p5; //赋值
//reset() 无参数时,不再管理资源,意味着count--,可能会释放资源
p3.reset();
cout << p2.use_count() << endl; //2
//reset(T* p),不再管理原资源,接管新的资源(裸指针)
cout << p4.use_count() << endl; //2
p4.reset(new T(30));
cout << p4.use_count() << endl; //1
//swap() 交换资源
p4->show(); //30
p4.swap(p5);
p4->show(); //20
std::swap(p4, p5); //效果与 p4.swap(p5)一样
//智能指针作为参数传递
print1(p4); //值传递
print2(p4); //引用传递
print(*p4);
cout << "-----\n";
//删除器:
shared_ptr<T> p11(new T[3], del_T_arr);
shared_ptr<T> p12(new T[3]{7,8,9}, [](T* p){delete[]p;});
T* pa = p12.get();
pa = pa + 2;
pa->show();
cout << "-----\n";
```

智能指针: weak_ptr

```
class Parent;
typedef std::shared_ptr<Parent> ParentPtr;
class Child;
typedef std::shared_ptr<Child> ChildPtr;
class Parent {
public:
    Parent() { cout << "Parent构造\n"; }
    ~Parent() { cout << "Parent析构\n"; }
    ChildPtr child;
};
class Child {
public:
    Child() { cout << "Child构造\n"; }
    ~Child() { cout << "Child析构\n"; }
    ParentPtr parent;
};
int main() {
    ParentPtr p = make_shared<Parent>();
    ChildPtr c = make_shared<Child>();
    p->child = c;
    c->parent = p;
    //观察运行结果。。。资源没有被释放
    cout<<"parent: "<<p.use_count()<<endl; //2
    cout<<"child: "<<c.use_count()<<endl; //2
    return 0;
}
```

循环引用

```
Parent构造
Child构造
parent: 2
child: 2
请按任意键继续
```

weak_ptr是一种不控制所指向对象生存期的智能指针。

它指向一个由**shared_ptr**管理的对象。

不增加和减少引用计数,也不会销毁管理的对象。

可能管理了一份资源,也可能没有管理一份资源;

外部有**shared_ptr**在管理时,管理的指针就是有效的;

外部没有**shared_ptr**在管理时,管理的指针就是无效的。

//weak_ptr用法(接口)

```
weak_ptr<int> wp1; //空指针
```

```
shared_ptr<int> sp1 = make_shared<int>(100);
```

```
cout << sp1.use_count() << endl; //1
```

```
wp1 = sp1; //不会增加 引用计数
```

```
cout << sp1.use_count() << endl; //1
```

```
weak_ptr<int> wp2(sp1); //用shared_ptr构造
```

```
//use_count()是weak_ptr对应的shared_ptr的引用计数
```

```
cout << wp2.use_count() << endl; //1
```

```
//expired():若use_count()==0返回true,否则false
```

```
if (!wp2.expired()) { //没有过期
```

```
    //lock():转为shared_ptr
```

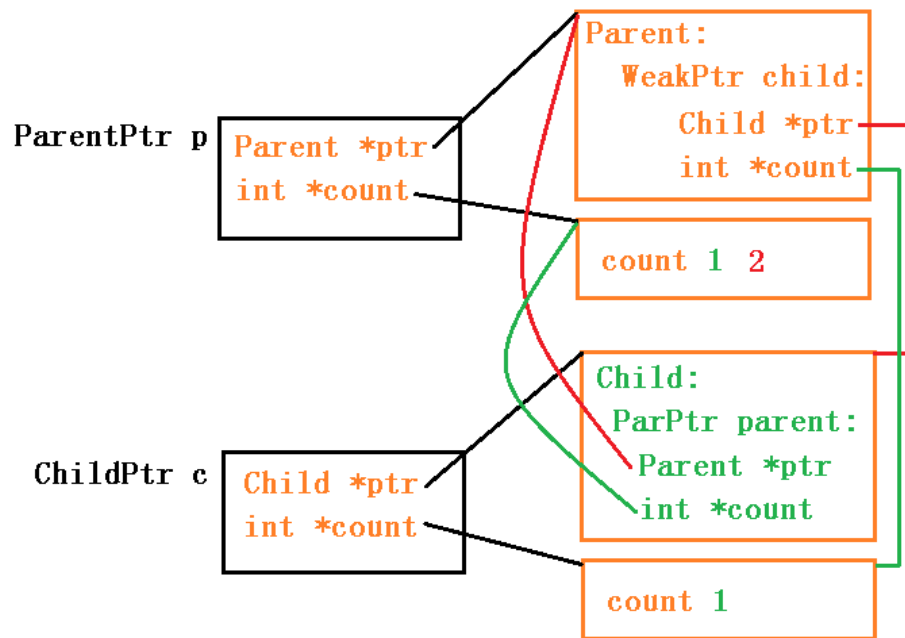
```
    shared_ptr<int> sp2 = wp2.lock();
```

```
    if (sp2) cout << "转换成功!\n";
```

```
}
```

智能指针: weak_ptr

```
#include <iostream>
#include <memory>
using namespace std;
class Parent;
typedef std::shared_ptr<Parent> ParentPtr;
class Child;
typedef std::shared_ptr<Child> ChildPtr;
typedef std::weak_ptr<Parent> WeakParentPtr;
typedef std::weak_ptr<Child> WeakChildPtr;
class Parent {
public:
    Parent() { cout << "Parent构造\n"; }
    ~Parent() { cout << "Parent析构\n"; }
    WeakChildPtr child;
};
class Child {
public:
    Child() { cout << "Child构造\n"; }
    ~Child() { cout << "Child析构\n"; }
    ParentPtr parent;
};
int main() {
    ParentPtr p = make_shared<Parent>();
    ChildPtr c = make_shared<Child>();
    p->child = c;
    c->parent = p;
    cout << "parent: " << p.use_count() << endl; //2
    cout << "child: " << c.use_count() << endl; //1
    return 0;
}
```



Parent和Child类中:
只要有一个修改为weak_ptr就可以了。

智能指针: unique_ptr

unique_ptr “拥有”它指向的对象。与shared_ptr不同，同一时刻只能有一个unique_ptr指向给定的对象。**没有引用计数**，不能普通拷贝和普通赋值，只能移动拷贝和移动赋值。

```
#include <iostream>
#include <memory>
#include <functional>
using namespace std;
class T {
public:
    T(int i = 0) :data(i) { cout << "T构造\n"; }
    ~T() { cout << "T析构\n"; }
    void show()const { cout << data << endl; }
private:
    int data;
};
//值传递: 拷贝
void print1(unique_ptr<T> p) {
    p->show();
}
//引用传递:
void print2(const unique_ptr<T> &p) {
    p->show();
}
void print(const T& obj) {
    cout << "对象的引用\n";
    obj.show();
}
void del_T_arr(T* parr) { //删除器
    delete[] parr;
}
```

```
//shared_ptr 用法(接口):
int main() {
    unique_ptr<T> p1; //空指针
    //explicit构造,只能直接初始化
    //unique_ptr<T> p2 = new T(100); //错误
    unique_ptr<T> p2(new T(100));
    //没有use_count(),unique(),因为没有引用计数
    (*p2).show(); //100 行为像指针 *
    p2->show(); //100 行为像指针 ->
    if (p2) cout << "p2 not null!\n";

    //没有普通拷贝构造函数
    //unique_ptr<T> p3(p2); //错误
    //unique_ptr<T> p3 = p2; //错误
    //没有普通赋值
    //p1 = p2; //错误

    //可以移动拷贝
    unique_ptr<T> p3 = std::move(p2);
    if (!p2) cout << "p2 empty\n";
    //可以移动赋值
    p1 = std::move(p3);

    T* p = p1.get(); //获取裸指针
    p->show(); //100
```

```
//release(),放弃控制权,返回指针
//p1.release(); //错,release不会释放资源
p = p1.release();
p->show(); //注意, p是裸指针,要delete才行

unique_ptr<T> p4(p); //重新接管

p4.reset(); //释放p4管理的资源
cout << "-----\n";

unique_ptr<T> p5(new T(200));
//先释放管理的资源,再接管新资源
p5.reset(new T(300));
cout << "=====\n";

//智能指针作为参数传递
print1(std::move(p5)); //值传递
unique_ptr<T> p6(new T(400));
print2(p6); //引用传递
print(*p6);
cout << "-----\n";

//删除器:
unique_ptr<T, function<void(T*)>> p11(new T[3],del_T_arr);
unique_ptr<T, function<void(T*)>> p12(new T[3]{ 7,8,9 },
    [](T* p)->void {delete[] p; });
cout << "=====\n";
return 0;
```

```
T构造
100
100
p2 not null!
p2 empty
100
100
T析构
-----
300
T析构
T构造
400
对象的引用
400
-----
T构造
T构造
T构造
T构造
T构造
T析构
T析构
T析构
T析构
T析构
T析构
请按任意键继续
```

智能指针: unique_ptr 简易实现

```
class T {
public:
    T(int i = 0) :data(i) { cout << "T构造\n"; }
    ~T() { cout << "T析构\n"; }
    void show()const { cout << data << endl; }
private:
    int data;
};

class Unique_ptr {
public:
    Unique_ptr():ptr(nullptr) { }
    explicit Unique_ptr(T* p):ptr(p){ }
    Unique_ptr(const Unique_ptr&) = delete;
    Unique_ptr& operator=(const Unique_ptr&) = delete;
    Unique_ptr(Unique_ptr&& other) noexcept//移动构造
        :ptr(other.ptr) {
        other.ptr = nullptr;
    }
    Unique_ptr& operator=(Unique_ptr&& other)noexcept{
        if (this == &other) return *this; //移动赋值
        del();
        std::swap(ptr, other.ptr);
        return *this;
    }
    ~Unique_ptr() { //析构
        del();
    }
};
```

```
public:
    T& operator*() {
        assert(ptr);
        return *ptr;
    }
    T* operator->() {
        assert(ptr);
        return ptr;
    }
    operator bool()const {
        return nullptr != ptr;
    }
    T* release() { //交出控制权
        T* p = ptr;
        ptr = nullptr;
        return p;
    }
    void reset(T* new_p=nullptr) { //重置
        del();
        ptr = new_p;
    }
    T* get() { return ptr; } //返回裸指针
private:
    void del(){delete ptr;ptr = nullptr;}
private:
    T* ptr; //管理的资源
};
```

```
int main() {
    Unique_ptr p1; //空指针
    //Unique_ptr p2 = new T(100); //错误
    Unique_ptr p2(new T(100));
    (*p2).show(); //100 行为像指针 *
    p2->show(); //100 行为像指针 ->
    if (p2) cout << "p2 not null!\n";
    //可以移动拷贝
    Unique_ptr p3 = std::move(p2);
    if (!p2) cout << "p2 empty\n";
    //可以移动赋值
    p1 = std::move(p3);
    T* p = p1.get(); //获取裸指针
    p->show(); //100
    p = p1.release();
    p->show(); //100 注意, p是裸指针
    Unique_ptr p4(p); //重新接管
    p4.reset(); //释放p4管理的资源
    cout << "-----\n";
    Unique_ptr p5(new T(200));
    //先释放管理的资源, 再接管新资源
    p5.reset(new T(300));
    cout << "=====\n";
    return 0;
}
```

```
T构造
100
100
p2 not null!
p2 empty
100
100
T析构
-----
T构造
T构造
T析构
-----
T析构
请按任意键继续
```

智能指针:小结

智能指针利用了RAII（资源分配即初始化）的技术对普通指针进行封装，这使得智能指针实质是一个对象，行为表现得却像一个指针。

智能指针的作用是防止忘记调用**delete**释放内存和程序异常忘记释放内存。另外指针的释放时机也是非常讲究的，多次释放同一个指针会造成程序崩溃，这些都可以通过智能指针来解决。

智能指针的使用也是有代价的，相比于裸指针，会有效率上的损失（空间和时间损失）

```
void fun(shared_ptr<int> ptr) { /*...*/ }
int main() {
    //1.一个裸指针不要用两个shared_ptr或unique_ptr来管理
    int *pi = new int(1); //少写这样的代码
    shared_ptr<int> pi1(pi);
    //shared_ptr<int> pi2(pi); //错
    shared_ptr<int> pi2 = pi1; //只能这样做

    //2.不要混用普通指针和智能指针
    shared_ptr<int> px1 = make_shared<int>(2);
    fun(px1); //ok,引用计数+1
    cout << *px1 << endl; //ok

    int *px = new int(2);
    //fun(px); //错,shared_ptr<int> ptr = px;不行
    fun(shared_ptr<int>(px)); //虽然可以运行,但是:
    cout << *px << endl; //此时 px已经被释放了!!

    //3.不要使用get初始化或者赋值给另一个智能指针
    shared_ptr<int> p(new int(3));
    int *p1 = p.get(); //获取p管理的裸指针,ok
    {
        //shared_ptr<int> p2(p1); //错,类似第1条
    }
    //4.不要delete get()返回的裸指针
    int *p2 = p.get();
    //delete p2; //错,会多次释放
    return 0;
}
```

运算符重载: new/delete

```
string *ps = new string("abc");  
string *ps_arr = new string[10];
```

new关键字:

第一步: new表达式调用了名为operator new(或者 operator new[])的标准库函数, 该函数分配一块足够大的、原始的、未命名的内存空间以便存储待定对象(或对象数组)。

第二步: 运行相应的构造函数以构造这些对象, 并为其传入初始值。

第三步: 对象被分配了空间并构造完成, 返回一个指向该对象的指针。

```
delete ps;  
delete [] ps_arr;
```

delete关键字:

第一步: 执行析构函数。

第二步: 调用operator delete(或者operator delete[])的标准库函数释放空间。

new/delete的执行流程不能更改, 但是可以重载operator new/operator delete;
可以通过定位new和显式调用析构函数来分离分配内存和构造析构对象。

运算符重载: new/delete

```
#include <iostream>
#include <cstdlib>
using namespace std;
void* operator new(std::size_t size) {
    if (void * p = malloc(size)) {
        cout << "my operator new\n";
        return p;
    }
    throw bad_alloc();
}
void operator delete(void* p) {
    cout << "my operator delete\n";
    free(p);
}
class A {
public:
    A(int i=0) :pi(new int(i)) {cout << "A构造\n";}
    ~A() { cout << "A析构!\n"; delete pi; }
    //隐式静态成员函数
    /*static*/void* operator new(std::size_t size){
        if (void * p = malloc(size)) {
            cout << "A operator new\n";
            return p;
        }
        throw bad_alloc(); //申请内存失败抛异常
    }
    /*static*/ void operator delete(void* p) {
        cout << "A operator delete\n";
        free(p);
    }
private:
    int* pi;
};
```

```
int main() {
    int *pi = new int(0);
    cout << "-----\n";
    delete pi;
    cout << "=====\n";
    A *pa = new A(1);
    cout << "-----\n";
    delete pa;
    return 0;
}
```



```
my operator new
-----
my operator delete
=====
A operator new
my operator new
A构造
-----
A析构!
my operator delete
A operator delete
请按任意键继续...
```

```
#include <iostream>
#include <cstdlib>
using namespace std;
class A {
public:
    A(int i=0) :pi(new int(i)) {cout << "A构造\n";}
    ~A() { cout << "A析构!\n"; delete pi; }
    //隐式静态成员函数
    /*static*/void* operator new[](std::size_t size){
        if (void * p = malloc(size)) {
            cout << "-A operator new[]\n";
            return p;
        }
        throw bad_alloc(); //申请内存失败抛异常
    }
    /*static*/ void operator delete[](void* p) {
        cout << "A operator delete[]\n";
        free(p);
    }
private:
    int* pi;
};
int main() {
    A *pa = new A[3];
    cout << "-----\n";
    delete[] pa;
    return 0;
}
```



```
-A operator new[]
A构造
A构造
A构造
-----
A析构!
A析构!
A析构!
A operator delete[]
请按任意键继续...
```

运算符重载: new/delete

```
#include <iostream>
#include <cstdlib>
using namespace std;
class A {
public:
    A(int i = 0) :pi(new int(i)) { cout << "A构造\n";}
    ~A() {cout << "A析构!\n"; delete pi; }
    static void* operator new(std::size_t size,
        const char* file, int line) {
        //跟踪在哪个文件哪行开辟的内存
        cout << __FILE__ << "----" << __LINE__ << endl;
        if (void * p = malloc(size)) {
            cout << "-A operator new\n";
            return p;
        }
        throw bad_alloc();
    }
private:
    int* pi;
};
int main() {
    A* pa = new(__FILE__, __LINE__) A(1);
    cout << "-----\n";
    delete pa;
    return 0;
}
```

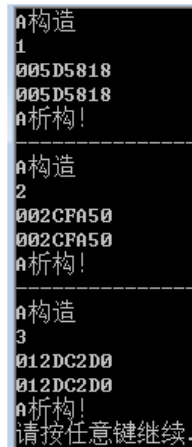
跟踪new的位置



```
j:\1m\t5\t5.cpp--11
-A operator new
A构造
-----
A析构!
请按任意键继续...
```

```
class A {
public:
    A(int i = 0) :pi(new int(i)) { cout << "A构造\n";}
    ~A() {cout << "A析构!\n"; delete pi; }
    void show()const { cout << *pi << endl; }
private:
    int* pi;
};
int main() {
    void* pbuf = malloc(100);
    //定位new,在 pbuf地址处:初始化一个对象
    //1.内存中 堆空间
    A *pa1 = new(pbuf) A(1);
    pa1->show(); //对象正确构造 1
    cout << pbuf << endl;
    cout << pa1 << endl;
    pa1->~A(); //显式调用析构
    free(pbuf);
    cout << "-----\n";
    char buf[200];
    //2.内存存在 栈空间
    A* pa2 = new(buf) A(2);
    pa2->show(); //对象正确构造 2
    cout << (void*)buf << endl;
    cout << pa2 << endl;
    // delete pa2; //错误,定位new出来的对象不能delete
    pa2->~A(); //显式调用析构
    return 0;
}
```

定位new,内存先准备好
在指定位置构造对象;
显式调用析构销毁对象。



```
A构造
1
005D5818
A析构!
-----
A构造
2
002CFA50
A析构!
-----
A构造
3
012DC2D0
A析构!
请按任意键继续...
```

std::allocator

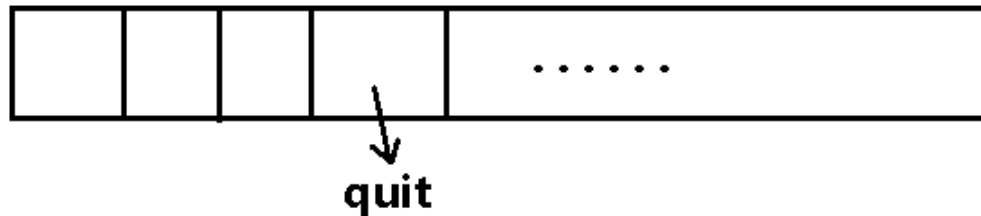
在vector等类型中，使用连续的内存存放元素。

使用内存分配和对象构造分离的方式，将会提供**更好的性能**和**更灵活的内存管理能力**。

标准库中包含一个名为**allocator**的类，允许我们将分配和初始化分离。头文件: <memory>

```
#include <iostream>
#include <string>
#include <utility>
using namespace std;

int main() {
    string *ps = new string[100];
    string s;
    string *q = ps;
    while (cin >> s && s != "quit"
           && q != ps + 100) {
        *q++ = std::move(s);
    }
    delete[] ps;
    return 0;
}
```



new string[100]，意味着要执行100次string类的默认构造函数，哪怕后面的用不到！

同样，析构也需要执行100次！

分配内存和构造分离：提高性能

销毁内存和析构分离：提高性能，内存可重用

std::allocator

```
#include <iostream>
#include <string>
#include <vector>
#include <memory>
using namespace std;

int main() {
    //定义一个allocator对象,为类型是string的对象分配内存
    allocator<string> alloc;
    //分配一段原始的、未构造的内存,可以保存10个string对象
    string * const p = alloc.allocate(10);
    string* q = p;
    alloc.construct(q++); //在q的位置构造对象(默认构造)
    alloc.construct(q++, 5, 'A'); // "AAAAA"
    alloc.construct(q++, "ABC"); // "ABC"
    //执行对象的析构函数
    while (q != p)
        alloc.destroy(--q);
    //释放一段内存, p是前面alloc.allocate()返回的指针,内存段中的对象
    //必须已经析构(已经调用过destroy())
    alloc.deallocate(p, 10);
}
```

```
vector<string> vec = {"abc", "xyz", "c++", "python"};
string * const p1 = alloc.allocate(vec.size()+3);
//通过拷贝vec中的元素来构造从 p1 开始的元素
string * q1 = uninitialized_copy(vec.begin(), vec.end(), p1);
//剩余元素初始化为 "hello"
q1 = uninitialized_fill_n(q1, 3, "hello");
string *q2 = p1;
while (q2 != q1)
    cout << *q2++<< "\t";
cout << endl;
return 0;
}
```

```
abc      xyz      c++      python  hello  hello  hello
请按任意键继续. . .
```

小练习

编写函数，返回一个动态分配的vector<int>,传入另一函数读取输入并存入vector,调用for_each按多种分割符打印

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
using namespace std;
vector<int>* new_vector() {
    return new (nothrow) vector<int>;
}
void input_vector(vector<int>* pv) {
    int v;
    while (cin >> v)
        pv->push_back(v);
}
ostream& print(ostream& out, const int& v, char sep) {
    return out << v << sep;
}
int main() {
    vector<int> *pv = new_vector();
    if (!pv) { cout << "内存不足!\n"; return -1; }
    input_vector(pv);
    //以 '\t' 作为分隔符打印输出
    for_each(pv->begin(), pv->end(),
        bind( [ ](int v) { return print(cout, v, '\t'); } ));
    cout << endl;
    delete pv;
    pv = nullptr;
    return 0;
}
// 答案: print, ref(cout), std::placeholders::_1, '\t'
```



```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
#include <memory>
using namespace std;
shared_ptr<vector<int>> new_vector() {
    return [ ]();
}
void input_vector(shared_ptr<vector<int>> pv) {
    //注意值传递和引用传递的区别
    cout << pv.use_count() << endl;
    int v;
    while (cin >> v)
        pv->push_back(v);
}
ostream& print(ostream& out, const int& v, char sep) {
    return out << v << sep;
}
int main() {
    shared_ptr<vector<int>> pv = new_vector();
    input_vector(pv);
    //以 ' ' 作为分隔符打印输出
    for_each(pv->begin(), pv->end(),
        bind(print, ref(cout), std::placeholders::_1, ' '));
    cout << endl;
    return 0;
}
// 答案: make_shared<vector<int>>()
```

