C++面筋

重载和重写的区别?

重载和重写的区别有以下几点：

一、定义上的区别：

- 如果同一作用域内的几个函数名字相同但形参列表不同，我们称之为函数重载（overlord）。
- 重写(override)是指在派生类中重新对基类中的虚函数（注意是虚函数）重新实现。即函数名和函数列表都一样，只是函数的实现体不一样。
- 隐藏：是指派生类的函数屏蔽了与其同名的基类函数，注意只要同名函数，不管参数列表是否相同，基类函数都会被隐藏。也指一个作用域隐藏了外部作用域的同名函数。

二、规则上的不同：

1、重载的规则：

①必须具有不同的参数列表。

②可以有不同的访问修饰符。

③可以抛出不同的异常。

2、重写方法的规则：

①参数列表必须完全与被重写的方法相同，否则不能称其为重写而是重载。

②返回的类型必须一直与被重写的方法的返回类型相同，否则不能称其为重写而是重载。

③访问修饰符的限制一定要大于被重写方法的访问修饰符。

④重写方法一定不能抛出新的检查异常或者比被重写方法申明更加宽泛的检查型异常。

三、类的关系上的区别：

重载和重写的区别：

（1）范围区别：重写和被重写的函数在不同的类中，重载和被重载的函数在同一类中。

（2）参数区别：重写与被重写的函数参数列表一定相同，重载和被重载的函数参数列表一定不同。

（3）virtual的区别：重写的基类必须要有virtual修饰，重载函数和被重载函数可以被virtual修饰，也可以没有。

隐藏和重写，重载的区别：

（1）与重载范围不同：隐藏函数和被隐藏函数在不同类中。

（2）参数的区别：隐藏函数和被隐藏函数参数列表可以相同，也可以不同，但函数名一定同；当参数不同时，无论基类中的函数是否被virtual修饰，基类函数都是被隐藏，而不是被重写。

这样理解就好了，基类的虚函数会带到子类中去（相当于又重新声明了一遍），所以就不会出现隐藏，而非虚函数就不同了，它没有重复声明。

参考

C++中重载、重写（覆盖）和隐藏的区别

H4 **C++内存布局**

答：

对于不是菱形继承的，可以看如下示例

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

class Base1 {
public:
    int a = 1;
    int b = 1;
    int c = 1;
};

class Base2 {
public:
    int a = 2;
    int b = 2;
    int c = 2;
};

class Derived: public Base1, public Base2 {
public:
    int a = 3;
};

int main() {
    cout << sizeof(Derived) << endl;
    Derived *p = new Derived();

    int *intp = reinterpret_cast<int*>(p);

    for (int i = 0; i < 7; i++) {
        cout << *(intp + i) << endl;
```

```
    }

    return 0;
}
```

输出如下

```
taojiandeMacBook-Pro:vfptrtest taojian$ ./a.out
28
1
1
1
2
2
2
3
taojiandeMacBook-Pro:vfptrtest taojian$ ls
```

可以看到c++对于继承的处理，先是基类Base1的变量，接着是基类Base2的变量，最后是自己定义的变量。

菱形继承

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

class A {
public:
    A(int a, int b) {
        this->a = a;
        this->b = b;
    }
    int a;
    int b;
};

class B1: public A {
public:
    B1(int a, int b):A(1,2) {
        this->a = 3;
        this->b = 4;
    }
    int a;
    int b;
};

class B2: public A {
public:
```

```
    B2(int a, int b):A(5,6) {
        this->a = 7;
        this->b = 8;
    }
    int a;
    int b;
};

class C: public B1, public B2 {
public:
    C(int a, int b):B1(1,1), B2(2,2) {
        this->a = 9;
        this->b = 10;
    }
    int a;
    int b;
};

int main() {
    int* p = reinterpret_cast<int*>(new C(1,1));
    for (int i = 0; i < 10; i++) {
        cout << p[i] << endl;
    }
    return 0;
}
```

```
(base) taojiandeMacBook-Pro:vfptrtest taojian$ ./a.out
1
2
3
4
5
6
7
8
9
10
(base) taojiandeMacBook-Pro:vfptrtest taojian$ cat b.cpp
```

其实还是直观明了的，虽然C两次间接继承了A，但是其实C对象中还是包含了两个A对象，互不打扰。

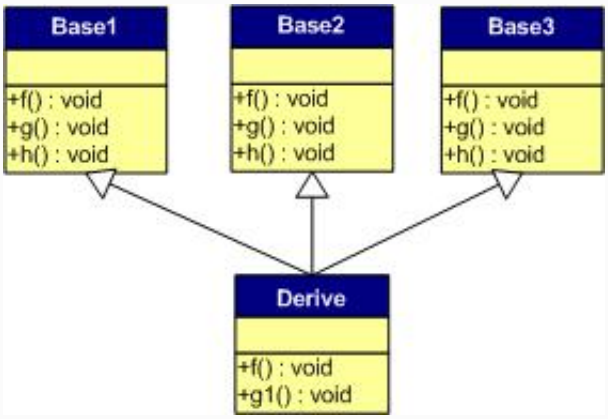
所以说，不管是菱形继承或者非菱形继承，都是依次铺开继承的对象，最后加上自己新定义的对象。

H4 C++虚函数表

只有一个类拥有虚函数时，它才有虚函数表指针。

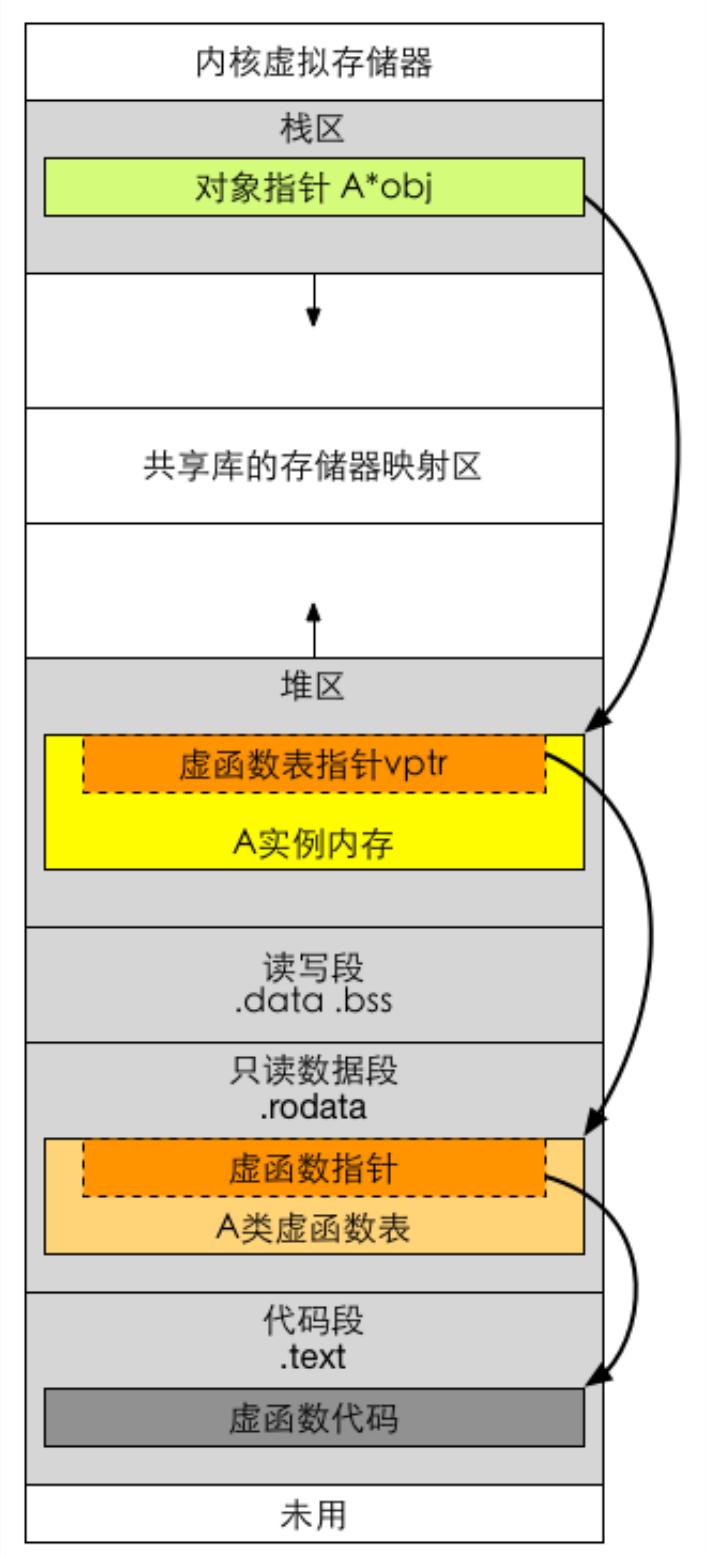
一个拥有虚函数的类对象，该对象存储的是虚函数表指针，虚函数表在只读区。

多继承情况



以上两图非常正确，我在linux和mac上测试了。从这幅图中可以看到，继承了几个虚类，那么就应该有几个虚函数指针。当子类中新增了虚函数，则虚函数指针跟在第一个虚函数表后面。

下图表示一个拥有虚函数的类对象的内存布局。obj是在函数中一个局部变量，它是一个对象指针，存储在栈中。它指向的对象在堆中。虚函数表在只读数据区。



H4 智能指针

智能指针是一个模版类。智能指针的使用方式与普通指针类似。解引用一个智能指针返回它指向的对象。

会有多个`shared_ptr`指向同一个对象，但是只能有一个`unique_ptr`指向一个对象。

`shared_ptr`和`unique_ptr`都支持的操作

```
shared_ptr<T>    sp;       // 空智能指针，可以指向类型为T的对象
unique_ptr<T>    up;
p;                         // 将p用作一个条件判断，若p指向一个对象，则为true
*p;                        // 解引用p，获得它指向的对象
p->mem;                    // 等价于(*p).mem
p.get();                   // 返回p中保存的指针。要小心使用，若智能指针释放了其对象，返回的指针所指向的对象也就消失了
swap(p, q);                // 交换p和q中的指针
p.swap(q)
```

`shared_ptr`独有的操作

```
make_shared<T>(args);    // 返回一个shared_ptr,指向一个动态分配的类型为T对象。使用args初始化对象

shared_ptr<T>p(q);       // p是shared_ptr q的拷贝；此操作会递增q中的计数器。q中的指针必须能转换为T*

p = q;                   // p和q都是shared_ptr，所保存的指针必须能相互转换。此操作会递减p的引用计数，递增q的引用计数；若p的引用计数变为0，则将其管理的原内存释放
p.unique();              // 若p.use_count()为1，返回true；否则返回false

p.use_count();           // 返回与p共享的智能指针数量，包括q；可能很慢，主要用于调试（为什么可能会很慢）
```

`unique_ptr`操作

```
unique_ptr<T> u1;        // 空unique_ptr,可以指向类型为T的对象。u1会调用delete来释放他的指针；

unique_ptr<T, D> u2;     // u2会使用一个类型为D的可调用对象来释放它的指针
unique_ptr<T, D> u(d);   // 空unique_ptr,指向类型为T的对象，用类型为D的对象d代替delete
u = p;                   // 出错，没有可用的重载符=
u = nullptr;             // 释放u指向的对象，将u置空
u.release();             // u放弃对指针的控制权，返回指针，并将u置为空
u.reset();               // 释放u指向的对象
u.reset(q);              // 如果提供了内置指针q，令u指向这个对象；否则将u只为空
u.reset(nullptr);        //
```

`weak_ptr`是一种不控制所指向对象生存期的智能指针，即`weak_ptr`不能向`shared_ptr`对所指向的对象进行操作。我猜测`weak_ptr`没有重载`.`和`->`运算符。它指向一个shared_ptr管理的对象

```
weak_ptr<T> w;                  // 空weak_ptr可以指向类型为T的对象
weak_ptr<T> w(sp);              // 与shared_ptr sp指向相同对象的weak_ptr。T必须能转换为sp指向的类型
w = p;                          // p可以是一个shared_ptr或一个weak_ptr。赋值后w与p共享对象
w.reset();                      // 将w置为空
w.use_count();                  // 与w共享对象的shared_ptr数量
w.expired();                    // 若w.user_count()为0，返回true。否则返回false
w.lock();                       // 如果expired为true，返回一个空shared_ptr；否则返回一个指向w的对象的shared_ptr
```

类型	解释
`shared_ptr`	共享指针
`unique_ptr`	强指针，独享
`weak_ptr`	弱指针，不能解引用

C++多态指的是什么，如何实现C++多态

答：在C++语言中，当我们使用基类的引用或指针调用一个虚函数时将发生动态绑定，虚函数会根据对象实际的类型执行不同版本的虚函数，这就是C++多台。C++使用虚函数以及虚函数表实现多态。

C++析构可不可以抛异常

1） C++中析构函数的执行不应该抛出异常； 2） 假如析构函数中抛出了异常，那么你的系统将变得非常危险，也许很长时间什么错误也不会发生；但也许你的系统有时就会莫名奇妙地崩溃而退出了，而且什么迹象也没有，崩得你满地找牙也很难发现问题究竟出现在什么地方； 3） 当在某一个析构函数中会有一些可能（哪怕是一点点可能）发生异常时，那么就必须要把这种可能发生的异常完全封装在析构函数内部，决不能让它抛出函数之外（这招简直是绝杀！呵呵！）； 4） 主人公阿愚吐血地提醒朋友们，一定要切记上面这几条总结，析构函数中抛出异常导致程序不明原因的崩溃是许多系统的致命内伤！

C++动态绑定和静态绑定

为了支持c++的多态性，才用了动态绑定和静态绑定。理解他们的区别有助于更好的理解多态性，以及在编程的过程中避免犯错误。 需要理解四个名词： 1、对象的静态类型：对象在声明时采用的类型。是在编译期确定的。 2、对象的动态类型：如果是指针，则是所指对象的类型，如果是引用，则为引用的类型。是在运行期决定的。对象的动态类型可以更改，但是静态类型无法更改。 关于对象的静态类型和动态类型，看一个示例：

```
class B
{
}
```

```
class C : public B
{
}
class D : public B
{
}
D* pD = new D();//pD的静态类型是它声明的类型D*，动态类型也是D*
B* pB = pD;//pB的静态类型是它声明的类型B*，动态类型是pB所指向的对象pD的类型D*
C* pC = new C();
pB = pC;//pB的动态类型是可以更改的，现在它的动态类型是C*
```

3、静态绑定：绑定的是对象的静态类型，某特性（比如函数）依赖于对象的静态类型，发生在编译期。 4、动态绑定：绑定的是对象的动态类型，某特性（比如函数）依赖于对象的动态类型，发生在运行期。

```
class B
{
    void DoSomething();
    virtual void vfun();
}
class C : public B
{
    void DoSomething();//首先说明一下，这个子类重新定义了父类的no-virtual函数，这是一个不好的设计，会导致名称遮掩；这里只是为了说明动态绑定和静态绑定才这样使用。
    virtual void vfun();
}
class D : public B
{
    void DoSomething();
    virtual void vfun();
}
D* pD = new D();
B* pB = pD;
```

让我们看一下，pD->DoSomething()和pB->DoSomething()调用的是同一个函数吗？ 不是的，虽然pD和pB都指向同一个对象。因为函数DoSomething是一个no-virtual函数，它是静态绑定的，也就是编译器会在编译期根据对象的静态类型来选择函数。pD的静态类型是D，*那么编译器在处理pD->DoSomething()的时候会将它指向D::DoSomething()。同理，pB的静态类型是B*，那pB->DoSomething()调用的就是B::DoSomething()。

让我们再来看一下，pD->vfun()和pB->vfun()调用的是同一个函数吗？ 是的。因为vfun是一个虚函数，它动态绑定的，也就是说它绑定的是对象的动态类型，pB和pD虽然静态类型不同，但是他们同时指向一个对象，他们的动态类型是相同的，都是D*，所以，他们的调用的是同一个函数：D::vfun()。

上面都是针对对象指针的情况，对于引用（reference）的情况同样适用。

指针和引用的动态类型和静态类型可能会不一致，但是对象的动态类型和静态类型是一致的。 D D; D.DoSomething()和D.vfun()永远调用的都是D::DoSomething()和D::vfun()。

至于哪些是动态绑定，哪些是静态绑定，有篇文章总结的非常好: 我总结了一句话：只有虚函数才使用的是动态绑定，其他的全部是静态绑定。目前我还没有发现不适用这句话的，如果有错误，希望你可以指出来。

特别需要注意的地方 当缺省参数和虚函数一起出现的时候情况有点复杂，极易出错。我们知道，虚函数是动态绑定的，但是为了执行效率，缺省参数是静态绑定的。

有上面的分析可知pD->vfun()和pB->vfun()调用都是函数D::vfun()，但是他们的缺省参数是多少？ 分析一下，缺省参数是静态绑定的，pD->vfun()时，pD的静态类型是D*，所以它的缺省参数应该是20；同理，pB->vfun()的缺省参数应该是10。编写代码验证了一下，正确。 对于这个特性，估计没有人会喜欢。所以，永远记住：“绝不重新定义继承而来的缺省参数（Never redefine function's inherited default parameters value.）”

关于c++语言 目前我基本上都是在c++的子集“面向对象编程”下工作，对于更复杂的知识了解的还不是很多。即便如此，到目前为止编程时需要注意的东西已经很多，而且后面可能还会继续增多，这也许是很多人反对c++的原因。 c++是Google的四大官方语言之一。但是Google近几年确推出了go语言，而且定位是和c/c++相似。考虑这种情况，我认为可能是Google的程序员们深感c++的复杂，所以想开发一种c++的替代语言。有时间要了解一下go语言，看它在类似c++的问题上时如何取舍的。

深入理解C++的动态绑定和静态绑定

H4 **c++中析构函数和构造函数都能不能是虚函数？为什么？**

当一个类有继承子类时，为了使得析构函数能够按照先子类后父类的顺序正确调用，应该把析构函数定义为虚函数。如果不是这样就会发生错误，举个例子如下:

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

class father {
public:
    ~father() {
        cout << "father deconstructor call" << endl;
    }
};

class son: public father {
public:
    ~son() {
        cout << "son deconstructor call" << endl;
    }
};

int main() {
    father* p = new son();
    delete p;
```

```
    return 0;
}
```

这样做只会调用基类的析构函数。如果把析构函数定义为虚函数，那么就能够正确调用析构函数，如下所示

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

class father {
public:
    virtual ~father() {
        cout << "father deconstructor call" << endl;
    }
};

class son: public father {
public:
    virtual ~son() {
        cout << "son deconstructor call" << endl;
    }
};

int main() {
    father* p = new son();
    delete p;

    return 0;
}
```

原因是因为是析构函数是虚函数，发生了动态绑定，delete p会执行p实际所指对象的析构函数。

C++析构函数为什么要为虚函数

构造函数不能是虚函数

从虚函数的实现方式即虚函数表来说是矛盾的

1.虚函数对应一个虚指针，虚指针其实是存储在对象的内存空间的。如果构造函数是虚的，就需要通过虚指针执行那个虚函数表（编译期间生成属于类）来调用，可是对象还没有实例化，也就是内存空间还没有，就没有虚函数表指针，所以构造函数不能是虚函数。

从语义上说是矛盾的

2.虚函数的作用在于通过父类的指针或者引用来调用它的时候能够变成调用子类的那个成员函数。而构造函数是在创建对象时自动调用的，不可能通过父类的指针或者引用去调用，因此也就规定构造函数不能是虚函数。

轮询，中断，DMA的各自特点和优缺点（轮训的优点我当时真的。。。。。）

说一下指针和引用?

1. 指针有自己的一块空间，而引用只是一个别名；
2. 使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小；
3. 指针可以被初始化为NULL，而引用必须被初始化且必须是一个已有对象的引用；
4. 作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引用的修改都会改变引用所指向的对象；
5. 指针在使用中可以指向其它对象，但是引用只能是一个对象的引用，不能 被改变；
6. 指针可以有多级指针（**p），而引用至于一级；
7. 指针和引用使用++运算符的意义不一样；
8. 如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。

C++指针和引用及区别

c++传参的几种方式?

值传递、指针传递和引用传递

举几个动态绑定和静态绑定的例子说明一下

参考c++动态绑定和静态绑定

返回一个临时的指针变量可不可以？为什么？从编译角度分析一下它的生命周期?

这要看指针指向的对象是什么，如果是new创建的就可以，如果指向栈空间就不可以。

指针的空间在哪分配?

指针的空间要看指针在哪里，指针变量是局部的还是全局的，或者是static。指针指向的对象空间在堆分配。

vector和list用法和实现方式?

vector的底层是一块连续存储区域，list底层是双向链表。

```
list常用用法
push_front, push_back, erase, pop_front, pop_back, clear, remove, unique,
splice, merge, reverse, sort
```

vector用的时候有没有什么需要注意的地方?

- [],begin(),end()是不进行范围检查的，如果发生了越界，将会产生未定义行为。
- 一旦内存重新配置，和元素相关的指针，引用和迭代器都将失效。
- 内存重新分配会很耗费时间。
-

使用vector需要注意的要点

H4 如果有一个1,2,3,4,5,6的vector删除奇数怎么做？（erase导致的指针指向变化）

H4 堆空间和栈空间?

堆空间是用户管理的，用户对栈空间是无感知的。

H4 static关键词的作用/内存位置

修饰什么	作用
修饰全局变量	隐藏全局变量的可见性
修饰局部变量	只初始化一次，此后一直保存，不会随着函数的退出就消失了
修饰成员变量	所有对象共享，独立与独享存在
修饰函数	隐藏可见性
修饰成员函数	由于static修饰的类成员属于类，不属于对象，因此static类成员函数是没有this指针的，this指针是指向本对象的指针。正因为没有this指针，所以static类成员函数不能访问非static的类成员，只能访问static修饰的类成员

c++ static类成员，static类成员函数

堆空间是动态分配的吗?

是的

static全局变量和普通全局变量的区别

全局变量(外部变量)的说明之前再冠以static 就构成了静态的全局变量。

全局变量本身就是静态存储方式， 静态全局变量当然也是静态存储方式。 这两者在存储方式上并无不同。

这两者的区别在于非静态全局变量的作用域是整个源程序， 当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。 而静态全局变量则限制了其作用域， 即只在定义该变量的源文件内有效， 在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其它源文件中引起错误。

static全局变量只初使化一次，防止在其他文件单元中被引用;

C语言中static全局变量与普通的全局变量区别

static局部变量和普通局部变量有什么区别？

把局部变量改变为静态变量后是改变了它的存储方式即改变了它的***生存期***。把全局变量改变为静态变量后是改变了它的***作用域***，限制了它的使用范围。

static局部变量只被初始化一次，下一次依据上一次结果值;

static函数与普通函数有什么区别?

static函数与普通函数作用域不同,仅在本文件。只在当前源文件中使用的函数应该说明为内部函数(static修饰的函数)，内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件.

static函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝

const

const引用

```
const int c = 1024;
const int &r1 = ci;
r1 = 42;              // 错误：r1是对常量的引用
int &r2 = ci;         // 错误：试图让一个非常量引用指向一个常量对象
```

第一个错误是显然的，因为r1是对常量的引用，自然r1也是不可修改的。

对于第二个错误，可以这样想，r2是对一个常量的引用，如果r2可以被修改，那么这个常量也被修改了。

const与指针

```
// 常量指针，p指向一个常量
const int *p = new int();
// *p = 2; 报错
p = new int();

// 指针常量，p是一个常量，不可以修改p的值，但是可以修改p指向的值
int * const p2 = new int;
// p2 = new int(); 报错
*p2 = 2;
```

顶层const和底层const

顶层const表示指针本身是一个常量，底层const表示指针所指对象是一个常量。

顶层const变量赋值没有影响，但底层const变量赋值就有影响了。不能把一个底层const变量赋值给一个底层非const变量，因为这样会造成其所指或所引用对象值的修改。

C++类中方法后面加const

表明方法只能读取类的成员变量值，不能修改类的成员变量值。

H4 在头文件里面声明一个static变量，在两个不同的cpp里面#include这个变量有没有问题

H4 vector迭代器失效问题

当vector内存重新分配时，会造成索引，begin，end，迭代器全部失效。当使用erase时，会造成删除元素后的迭代器，索引，end失效。

H4 有哪些构造函数？什么情况下会用到构造函数？

类的**构造函数**是类的一种特殊的成员函数，它会在每次创建类的新对象时执行。

```
class car {
public:
    // 默认构造函数，当且仅当没有定义任何构造函数时，编译器才提供默认构造函数
    car() {}
    //
};
```

H4 C++空类中，默认提供了哪几种成员函数

默认构造函数，赋值构造函数，赋值操作赋，默认析构函数，地址操作符

```
class Empty   {
public:
        Empty(); // 缺省构造函数
        Empty( const Empty& ); // 拷贝构造函数

        ~Empty(); // 析构函数

        Empty& operator=( const Empty& ); // 赋值运算符

        Empty* operator&(); // 取址运算符
        const Empty* operator&() const; // 取址运算符 const

  };
```

默认构造函数

如果类没有定义任何构造函数，则编译器自动生成一个默认构造函数。默认构造函数可以有参数，只需要所有参数都有默认值即可。

```
class test{

};

int main() {
    test a;
    test a = test();
}
```

复制构造函数

复制构造函数用于将一个对象复制到新创建的对象中。也就是说，它用于初始化过程中，而不是常规的赋值过程中。类的肤质构造函数原型通常如下

`Class_name(const Class_name&);`

新建一个对象并将其初始化为同类现有对象时，复制构造函数都将被调用。下面四种声明都将调用复制构造函数

```
StringBad ditto(motto);
StringBad metoo = motoo;
StringBad also = StringBad(motto);
StringBad * pStringBad = new StringBad(const StringBad &);
```

其中中间的2种声明可能会使用复制构造函数直接创建metoo和also，也可能使用复制构造函数生成一个临时对象，然后将临时对象的内容赋给metoo和also，这取决于具体的实现。

每当程序生成了对象副本时，编译器都将使用复制构造函数。具体地说，当函数按值传递对象或函数返回对象时，都将使用复制构造函数。无论哪种编译器，当按值传递和返回对象时，都将调用复制构造函数。

复制构造函数的功能

默认的复制构造函数逐个复制非静态成员（成员复制也称为浅复制），复制的是成员的值。所以说如果成员变量是指针的话，就容易出现问题。如果是共享指针的话可以解决这个问题。

赋值操作赋

看下面一段代码

```
#include <iostream>
using namespace std;
class car {
public:
    car() {
        cout << "default constructor" << endl;
    }
    // 接受一个参数的构造函数允许使用赋值句法来将对象初始化为一个值。
    car (int x) {
        cout << "one variable constructor" << endl;
        this->a = x;
    }
    ~car() {
        cout << "deconstructor called" << endl;
    }
    int getA() {
        return a;
    }

    car& operator = (const car &b) {
        this->a = b.a;
        cout << "赋值操作赋调用" << endl;
        return *this;
    }
private:
    int a;
};

int main() {
    car a = car();
    // 先调用一个变量构造函数，创建一个局部变量，再调用赋值操作符将局部变量值赋值给a。然后调用析构函数将临时变量析构掉。
    a = 2;
    return 0;
}
```

```
default constructor
one variable constructor
赋值操作赋调用
deconstructor called
deconstructor called
Program ended with exit code: 0
```

与复制构造函数相似，赋值操作符的隐式实现也对成员逐个复制。如果成员本身就是类对象，则程序将使用为这个类定义的赋值操作符来复制该成员，但静态数据成员不受影响。

new和malloc的区别

- new即是操作符又是关键字，malloc是c库函数
- new作为关键字，即new表达式时，底层会调用malloc函数。new表达式先调用operator new的库函数，operator new会调用malloc。接着会调用类的构造函数。最后返回类的指针。

c++ new 与malloc有什么区别

new, operator new ,placement new

new 过程

当我们使用一条new表达式时：

```
// new 表达式
string *sp = new string("a value");
string *arr = new string[10];
```

实际执行了三步操作：第一步，new表达式调用一个名为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特定类型的对象（或者对象的数组）。第二步：编译器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 过程

当我们使用一条delete表达式删除一个动态分配的对象时：

```
delete sp;          // 销毁*sp，然后释放sp指向的内存空间
delete [] arr;      //  销毁数组中的元素，然后释放对应的内存空间
```

实际执行了两步操作：第一步，对sp所指的对象或者arr所指的数组中的元素执行对应的析构函数。第二步，编译器调用名为operator delete（或者operator delete []）的标准库函数释放内存空间。

operator new函数

（1）只分配所要求的空间，不调用相关对象的构造函数。当无法满足所要求分配的空间时，则 ->如果有new_handler，则调用new_handler，否则 ->如果没要求不抛出异常（以nothrow参数表达），则执行bad_alloc异常，否则 ->返回0 （2）可以被重载 （3）重载时，返回类型必须声明为void* （4）重载时，第一个参数类型必须为表达要求分配空间的大小（字节），类型为size_t （5）重载时，可以带其它参数

new 、operator new 和 placement new 区别

（1）new ： 不能被重载，其行为总是一致的。它先调用operator new分配内存，然后调用构造函数初始化那段内存。

new 操作符的执行过程：

1. 调用operator new分配内存；
2. 调用构造函数生成类对象；
3. 返回相应指针

（2）operator new：要实现不同的内存分配行为，应该重载operator new，而不是new。

operator new就像operator + 一样，是可以重载的。如果类中没有重载operator new，那么调用的就是全局的::operator new来完成堆的分配。同理，operator new[]、operator delete、operator delete[]也是可以重载的。

（3）placement new：只是operator new重载的一个版本。它并不分配内存，只是返回指向已经分配好的某段内存的一个指针。因此不能删除它，但需要调用对象的析构函数。

如果你想在已经分配的内存中创建一个对象，使用new时行不通的。也就是说placement new允许你在一个已经分配好的内存中（栈或者堆中）构造一个新的对象。原型中void* p实际上就是指向一个已经分配好的内存缓冲区的的首地址。

- C++Primer
- C++中的new、operator new与placement new

H4 **sizeof的实现原理，是需要函数支持呢，还是操作系统支持**

首先sizeof时关键字。是需要编译器支持的。不需要函数支持，也不需要操作系统支持。

```
#include <stdio.h>
#include <stdlib.h>

void my_print(int x) {
    printf("%d", x);
    return;
}

int main() {
    int x;
    scanf("%d", &x);
    int arr[x];
    int y = x + 255;

    print(y);

    return 0;
}
```

编译后得到汇编代码

```
    .file   "main.c"
    .text
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "%d"
    .text
    .globl  my_print
    .type   my_print, @function
my_print:
.LFB41:
    .cfi_startproc
    subq    $8, %rsp
    .cfi_def_cfa_offset 16
    movl    %edi, %edx
    leaq    .LC0(%rip), %rsi
    movl    $1, %edi
    movl    $0, %eax
    call    __printf_chk@PLT
    addq    $8, %rsp
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
.LFE41:
    .size   my_print, .-my_print
    .globl  main
    .type   main, @function
main:
.LFB42:
    .cfi_startproc
    subq    $24, %rsp
    .cfi_def_cfa_offset 32
    movq    %fs:40, %rax
    movq    %rax, 8(%rsp)
    xorl    %eax, %eax
    leaq    4(%rsp), %rsi
    leaq    .LC0(%rip), %rdi
    call    __isoc99_scanf@PLT
    movl    4(%rsp), %eax
    leal    255(%rax), %edi // %edi是y的值
    movl    $0, %eax
    call    print@PLT
    movq    8(%rsp), %rdx
    xorq    %fs:40, %rdx
    jne .L6
    movl    $0, %eax
    addq    $24, %rsp
```

```
    .cfi_remember_state
    .cfi_def_cfa_offset 8
    ret
.L6:
    .cfi_restore_state
    call    __stack_chk_fail@PLT
    .cfi_endproc
.LFE42:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
    .section    .note.GNU-stack,"",@progbits
```

参考

sizeof的结果是在哪个阶段确定的

- 在编译阶段
- 在运行时阶段

让你实现sizeof的话怎么实现呢

- 基本类型
- 对象，要考虑字节对齐
- 数组，对象的大小乘以数组的长度

对象的首地址好确定，怎么确定一个对象的尾地址呢

对象首地址+sizeof(对象)

深浅拷贝

先考虑一种情况，对一个已知对象进行拷贝，编译系统会自动调用一种构造函数--拷贝构造函数，如果用户未定义拷贝构造函数，则会调用默认拷贝构造函数。

先看一个例子，有一个学生类，数据成员时学生的人数和名字：

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

class Student
{
private:
    int num;
    char *name;
public:
    Student();
    ~Student();
};

Student::Student()
{
```

```
    name = new char(20);
    cout << "Student" << endl;

}
Student::~Student()
{
    cout << "~Student " << (long long)name << endl;
    delete name;
    name = NULL;
}

int main()
{
    {// 花括号让s1和s2变成局部对象，方便测试
        Student s1;
        Student s2(s1);// 复制对象
    }

    return 0;
}
```

调用一次默认构造函数，调用一次默认拷贝构造函数，而执行两次析构函数。由于默认拷贝构造函数执行的是前拷贝操作，所以s1和s2中name执行同一块内存区域，当执行第二次析构函数时，delete name会释放掉之前已经被操作系统回收的内存（这个说法不严谨，还需要查证）。

所以，在对含有指针成员的对象进行拷贝时，必须要自己定义拷贝构造函数，使拷贝后的对象指针成员有自己的内存空间，即进行深拷贝，这样就避免了内存泄漏发生。

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

class Student
{
private:
    int num;
    char *name;
public:
    Student();
    ~Student();
    Student(const Student &s);//拷贝构造函数，const防止对象被改变
};

Student::Student()
{
    name = new char(20);
```

```
    cout << "Student" << endl;

}

Student::~Student()
{
    cout << "~Student " << (long long)name << endl;
    delete name;
    name = NULL;
}

Student::Student(const Student &s)
{
    name = new char(20);
    memcpy(name, s.name, strlen(s.name));
    cout << "copy Student" << endl;
}

int main()
{
    {// 花括号让s1和s2变成局部对象，方便测试
        Student s1;
        Student s2(s1);// 复制对象
    }
    system("pause");
    return 0;
}
```

当自己定义了拷贝构造函数后，执行深拷贝，那么此时就不会发生问题了。

C++内存存储区，static变量在哪个区域，生命周期

如果已经初始化，则放在.data，如果未初始化，则放在.bss。生命周期，一直存在。

- 代码段.txt

  .txt段存放代码（如函数）与部分整数常量（比如说立即数），.txt段的数据可以被执行。

- 数据段(.data)

  .data用于存放初始化过的全局变量。若全局变量值为0，为了优化编译器会将它放在.bss段中。

- bss段(.bss)

  .bss段被用来存放那些没有被初始过或者初始化为0的全局变量。bss段只占运行时的内存空间而不占文件空间。在程序运行的整个周期，.bss段的数据一直存在。

- 只读数据段(.rodata)

  ro表read only，用于存放不可变修改的常量数据，一旦程序中对其修改将会出现段错误： (1) 程序中的常量不一定就放在rodata中，有的立即数和指令编码放在.text中 (2) 对于字符串常量，若程序中存在重复的字符串，编译器会保证只存在一个 (3) rodata是在多个进程间共享的 (4) 有的嵌入式系统，rodata放在ROM(或者NOR FLASH)中，运行时直接读取无需加载至RAM(哈佛和冯

诺依曼，从STM32的const全局变量说起有所记录) 想要将数据放在.rodata只需要加上const属性修饰即可。

参考

如何避免隐式转换?

答：在构造函数前面加关键字explicit。

写一个函数指针

```
#include <iostream>
#include <string>
#include <memory>
#include <vector>

using namespace std;


bool compare(int x, int  y, bool (*p)(int x, int y)) {
    return p(x, y);
}

bool fun(int x, int y) {
    return x < y;
}

int main() {
    cout << compare(1, 3, fun) << endl;

    return 0;
}
```

内存泄漏平时用什么方法检测

内存泄漏及简单检测的一种方法

空类的大小是多少，如果是派生类呢

在c++中struct和class中，如果没有任何非静态成员变量，其对象会占用一个字节。

如果是派生类，还是会占用一个字节。

c++标准规定必须可以区分，c++标准禁止对象大小为0，因为两个不同的对象需要两个不同的地址表示。

C++ vector和list的区别?

vector是动态数组实现的，一说到动态那肯定是在堆上分配空间的。如果容量超出原先设定的值，会以2倍扩增。性能上：因为是数组实现的，所以访问起来肯定是O(1)时间内访问。

因为是vector,所以会经常有插入和删除的操作：

如果在结尾插入并且空间够的情况下，很快，如果空间不够，则首先要进行扩容，扩容的过程中完成内存拷贝。在中间拷贝也是一样，如果空间足够大，只需要完成插入位置后的元素拷贝就行了，如果内存不够则也需要先进行扩容，然后进行拷贝。

如果删除的是结尾的元素的话很快就可以完成，如果是中间的元素那就需要拷贝了。

总体而言由于vector的特性原因，所以它很适合随机访问，并且插入删除在结尾部。

list是双向链表实现的，由于是双向链表，所以肯定也是在堆上分配空间的。

那自然插入和删除都是很容易的，因为双向链表实现的原理就是为了插入和删除。

具体的区别和联系：

都是在堆上分配空间

vector是基于动态数组实现的，list基于双向链表实现的

vector不便于中间插入和删除，list支持随机插入和删除

执行main函数之前和之后做了哪些工作?

main函数执行之前主要是系统的初始化资源：

- 在栈区：设置栈指针
- 在data段：初始化全局变量和静态变量
- 在bss段：对未初始化的全局变量进行赋初值，bool是false,short,int,long 是0，指针是NULL
- 将main函数的参数传到main函数里面

main函数执行完成之后并不一定意味着进程结束。

main函数执行完成之后：

- 全局对象的析构函数会在main函数的执行后执行
- 使用atexit注册的函数会在main函数执行之后执行

```
#include <stdio.h>
#include <stdlib.h>

void fn1(void)
{
    printf("next.\n");
}

void fn2(void)
{
    printf("executed ");
}
```

```c
void fn3(void)
{
    printf("is ");
}

void fn4(void)
{
    printf("This ");
}

int main(void)
{
    // 注册需要在 main 函数结束后执行的函数.
    // 请注意它们的注册顺序和执行顺序
    // 在 main 函数结束后被调用，调用顺序与注册顺序相反。 先注册后执行。

    atexit(fn1);
    atexit(fn2);
    atexit(fn3);
    atexit(fn4);

    // 这条输出语句具有参照性，它可不是最后一句输出.
    puts("This is executed first.");

    // EXIT_SUCCESS 代表 0，它定义在 stdlib.h 中.
    // 我只是顺便提一下，也许你知道，但我担心你不知道，呵呵.
    return EXIT_SUCCESS;
}
```

你在写程序的时候如果程序出现了死循环你怎么找到这个死循环?

首先需要找可能出现死循环的进程，一个程序执行好久没有结果有可能有两个结果，第一：程序正在正常运行，但还没结果，第二：程序出现死循环 首先查看进程使用资源情况，如果内存占用正常，但是CPU占比接近100%,就说明可能出现死循环。 再使用pstack $pid查看进程栈，如果进程栈总是停留在一个位置，那这个位置就是死循环的位置，在文件里查看具体的代码就可以了。

C语言中extern

命名的强制类型转换

名称	解释	
`static_cast`	任何具有明确定义的类型转换，只要不包含底层const	
`const_static`	去掉底层const	
`reinterpret_cast`	运算对象的位模式提供较低层次上的重新解释	
`dynamic_cast`	运行时类识别	

H4 dynamic_cast怎么实现的

dynamic_cast运算符的主要用途：将基类的指针或引用安全地转换成派生类的指针或引用，并用派生类的指针或引用调用非虚函数。如果是基类指针或引用调用的是虚函数无需转换就能在运行时调用派生类的虚函数。

前提条件：当我们将dynamic_cast用于某种类型的指针或引用时，只有该类型至少含有虚函数时(最简单是基类析构函数为虚函数)，才能进行这种转换。否则，编译器会报错。即基类中至少有一个虚函数。

原理可以看下面这个链接

C++ dynamic_cast实现原理

C++中深入理解dynamic_cast

H4 什么时候用指针？引用?

严格来说bai，C++是不建议使用指针的，因为面向对象是引用和智能指zhi针的天下，因此，C++来说最好不要用指针而使用引用。但是有迫不得已的时候比如在类中定义一个指向成员变量的指针，在函数中进行动态申请的情况，这个时候智能使用指针，或者一些智能指针什么的。所以具体情况具体分析，没有什么绝对。建议尽可能少，因为每一个指针的存在，都可能需要内存管理和释放。

H4 include“my.h”与include<my.h>有什么区别

include <myheadfile.h> 代表编译时直接在软件bai设置指定的路径中du寻找里面是否有myheadfile.h文件。如果有，zhi直接加载dao；如果没有，报错。 include "myheadfile.h" 代表编译时先寻找你正在编辑的源代码文件(C或CPP文件)所在的文件夹里面有没有myheadfile.h文件。如果有，优先加载这个文件，如果没有，就会在软件设置指定的路径中寻找里面是否有myheadfile.h文件。如果有，直接加载；如果没有，报错。

H4 内存管理与内存分区

H4 什么是左值，什么是右值

C++对于左值和右值没有标准定义，但是有一个被广泛认同的说法：

- 可以取地址的，有名字的，非临时的就是左值；
- 不能取地址的，没有名字的，临时的就是右值；

可见立即数，函数返回的值等都是右值；而非匿名对象(包括变量)，函数返回的引用，const对象等都是左值。

从本质上理解，创建和销毁由编译器幕后控制，程序员只能确保在本行代码有效的，就是右值(包括立即数)；而用户创建的，通过作用域规则可知其生存期的，就是左值(包括函数返回的局部变量的引用以及const对象)。

定义右值引用的格式如下：

```
类型 && 引用名 = 右值表达式;
```

c++ 左值引用与右值引用

左值引用和右值引用

c++ 左值引用与右值引用

线程池是怎么实现的，有什么好处

26.将一个函数重载时只在原有基础上把参数修改为默认传参,可以编译通过吗?

答：可以。但是默认参数后面应该没有非默认参数。

5.C语言中free和delete区别?

答：1、new/delete是C++的操作符，而malloc/free是C中的函数。 2、new做两件事，一是分配内存，二是调用类的构造函数；同样，delete会调用类的析构函数和释放内存。而malloc和free只是分配和释放内存。 3、new建立的是一个对象，而malloc分配的是一块内存；new建立的对象可以用成员函数访问，不要直接访问它的地址空间；malloc分配的是一块内存区域，用指针访问，可以在里面移动指针；new出来的指针是带有类型信息的，而malloc返回的是void指针。 4、new/delete是保留字，不需要头文件支持；malloc/free需要头文件库函数支持。

参考

- 许愿一波腾讯offer 顺便分享下面筋
- 秋招结束，整理一下。内含面筋(后台/c++)