基础语言

参考网址

static关键字的作用

c语言中

静态局部变量: 使用static修饰符定义，即使在声明时未赋初值，编译器也会把它初始化为0。且静态局部变量存储于进程的全局数据区，即使函数返回，它的值也会保持不变。

> 变量在全局数据区分配内存空间；编译器自动对其初始化 其作用域为局部作用域，当定义它的函数结束时，其作用域随之结束

静态全局变量:仅对当前文件可见，其他文件不可访问，其他文件可以定义与其同名的变量，两者互不影响。

> 在定义不需要与其他文件共享的全局变量时，加上static关键字能够有效地降低程序模块之间的耦合，避免不同文件同名变量的冲突，且不会误使用。

函数:函数的使用方式与全局变量类似，在函数的返回类型前加上static，就是静态函数。其特性如下：

- 静态函数只能在声明它的文件中可见，其他文件不能引用该函数
- 不同的文件可以使用相同名字的静态函数，互不影响

c++面向对象

静态数据成员: 在类内数据成员的声明前加上static关键字，该数据成员就是类内的静态数据成员。其特点如下：

- 静态数据成员存储在全局数据区，静态数据成员在定义时分配存储空间，所以不能在类声明中定义
- 静态数据成员是类的成员，无论定义了多少个类的对象，静态数据成员的拷贝只有一个，且对该类的所有对象可见。也就是说任一对象都可以对静态数据成员进行操作。而对于非静态数据成员，每个对象都有自己的一份拷贝。
- 由于上面的原因，静态数据成员不属于任何对象，在没有类的实例时其作用域就可见，在没有任何对象时，就可以进行操作
- 和普通数据成员一样，静态数据成员也遵从`public, protected, private`访问规则
- 静态数据成员的初始化格式：`<数据类型><类名>::<静态数据成员名>=<值>`
- 类的静态数据成员有两种访问方式：`<类对象名>.<静态数据成员名> 或 <类类型名>::<静态数据成员名>`

静态成员函数 与静态数据成员类似，静态成员函数属于整个类，而不是某一个对象，其特性如下：

- 静态成员函数没有this指针，它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，它只能调用其余的静态成员函数
- 出现在类体外的函数定义不能指定关键字static
- 非静态成员函数可以任意地访问静态成员函数和静态数据成员

说一下c++和c的区别

设计思想：

C++是面向对象的语言，而C是面向过程的结构化编程语言

语法上：

C++具有重载、继承和多态三种特性

C++相比C，增加多许多类型安全的功能，比如强制类型转换、

C++支持范式编程，比如模板类、函数模板等

增加了引用行变量

扩充上：

增加了新的关键字和类型，如bool

说一下C++中static关键字的作用

- 对于函数定义和代码块之外的变量声明，static修改标识符的链接属性，由默认的external变为internal，作用域和存储类型不改变，这些符号只能在声明它们的源文件中访问。
- 对于代码块内部的变量声明，static修改标识符的存储类型，由自动变量改为静态变量，作用域和链接属性不变。这种变量在程序执行之前就创建，在程序执行的整个周期都存在。
- 对于被static修饰的普通函数，其只能在定义它的源文件中使用，不能在其他源文件中被引用
- 对于被static修饰的类成员变量和成员函数，它们是属于类的，而不是某个对象，所有对象共享一个静态成员。静态成员通过<类名>::<静态成员>来使用。

说一说c++中四种cast转换

C++中四种类型转换是：static_cast, dynamic_cast, const_cast, reinterpret_cast

1、const_cast

用于将const变量转为非const

2、static_cast

用于各种隐式转换，比如非const转const，void*转指针等, static_cast能用于多态向上转化，如果向下转能成功但是不安全，结果未知；

3、dynamic_cast

用于动态类型转换。只能用于含有虚函数的类，用于类层次间的向上和向下转化。只能转指针或引用。向下转化时，如果是非法的对于指针返回NULL，对于引用抛异常。要深入了解内部转换的原理。

向上转换：指的是子类向基类的转换

向下转换：指的是基类向子类的转换

它通过判断在执行到该语句的时候变量的运行时类型和要转换的类型是否相同来判断是否能够进行向下转换。

4、reinterpret_cast

几乎什么都可以转，比如将int转指针，可能会出问题，尽量少用；

5、为什么不使用C的强制转换？

C的强制转换表面上看起来功能强大什么都能转，但是转化不够明确，不能进行错误检查，容易出错。

请说一下C/C++ 中指针和引用的区别?

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 (针对不同的系统，指针大小不同)，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.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；

5.指针可以有多级指针（**p），而引用至于一级；

6.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；

7.可以有const指针，但是没有const引用；

8.指针和引用使用++运算符的意义不一样；

9.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。

10.不能建立数组的引用，指针是可以代表数组的，指向数组的指针是数组的首地址，但是引用是不可以的。

给定三角形ABC和一点P(x,y,z)，判断点P是否在ABC内，给出思路并手写代码

根据面积法，如果P在三角形ABC内，那么三角形ABP的面积+三角形BCP的面积+三角形ACP的面积应该等于三角形ABC的面积。算法如下：

```
#include <iostream>
#include <math.h>
using namespace std;
#define ABS_FLOAT_0 0.0001
struct point_float
{
    float x;
    float y;
};

/**

* @brief 计算三角形面积

*/

float GetTriangleSquar(const point_float pt0, const point_float pt1, const point_float pt2)
{
    point_float AB,   BC;
    AB.x = pt1.x - pt0.x;
    AB.y = pt1.y - pt0.y;
```

```
    BC.x = pt2.x - pt1.x;
    BC.y = pt2.y - pt1.y;
    return fabs((AB.x * BC.y - AB.y * BC.x)) / 2.0f;
}

/**

* @brief 判断给定一点是否在三角形内或边上

*/


bool IsInTriangle(const point_float A, const point_float B, const point_float C, const
point_float D)
{
    float SABC, SADB, SBDC, SADC;
    SABC = GetTriangleSquar(A, B, C);
    SADB = GetTriangleSquar(A, D, B);
    SBDC = GetTriangleSquar(B, D, C);
    SADC = GetTriangleSquar(A, D, C);
    float SumSuqar = SADB + SBDC + SADC;
    if ((-ABS_FLOAT_0 < (SABC - SumSuqar)) && ((SABC - SumSuqar) < ABS_FLOAT_0)){
        return true;
    } else {
     return false;
    }
}
```

请你说一下你理解的c++中的smart pointer四个智能指针:shared_ptr,unique_ptr,weak_ptr,auto_ptr

C++里面的四个智能指针: auto_ptr, shared_ptr, weak_ptr, unique_ptr 其中后三个是c++11支持，并且第一个已经被11弃用。 为什么要使用智能指针：

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

1. auto_ptr（c++98的方案，cpp11已经抛弃）

采用所有权模式。

```
auto_ptr< string> p1 (new string ("I reigned lonely as a cloud.”));
auto_ptr<string> p2;
p2 = p1; //auto_ptr不会报错.
```

此时不会报错，p2剥夺了p1的所有权，但是当程序运行时访问p1将会报错。所以auto_ptr的缺点是：存在潜在的内存崩溃问题！

2. unique_ptr（替换auto_ptr）

unique_ptr实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露(例如“以new创建对象后因为发生异常而忘记调用delete”)特别有用。

采用所有权模式，还是上面那个例子

```
unique_ptr<string> p3 (new string ("auto"));   //#4
unique_ptr<string> p4;                         //#5
p4 = p3;//此时会报错！！
```

编译器认为p4=p3非法，避免了p3不再指向有效数据的问题。因此，unique_ptr比auto_ptr更安全。

另外unique_ptr还有更聪明的地方：当程序试图将一个 unique_ptr 赋值给另一个时，如果源 unique_ptr 是个临时右值，编译器允许这么做；如果源 unique_ptr 将存在一段时间，编译器将禁止这么做，比如：

```
unique_ptr<string> pu1(new string ("hello world"));
unique_ptr<string> pu2;
pu2 = pu1;                                     // #1 not allowed
unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string ("You"));   // #2 allowed
```

其中#1留下悬挂的unique_ptr(pu1)，这可能导致危害。而#2不会留下悬挂的unique_ptr，因为它调用 unique_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而已的行为表明，unique_ptr 优于允许两种赋值的auto_ptr 。

注：如果确实想执行类似与#1的操作，要安全的重用这种指针，可给它赋新值。C++有一个标准库函数std::move()，让你能够将一个unique_ptr赋给另一个。例如：

```
unique_ptr<string> ps1, ps2;
ps1 = demo("hello");
ps2 = move(ps1);
ps1 = demo("alexia");
cout << *ps2 << *ps1 << endl;
```

3. shared_ptr

shared_ptr实现共享式拥有概念。多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。从名字share就可以看出了资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享。可以通过成员函数use_count()来查看资源的所有者个数。除了可以通过new来构造，还可以通过传入auto_ptr, unique_ptr,weak_ptr来构造。当我们调用release()时，当前指针会释放资源所有权，计数减一。当计数等于0时，资源会被释放。

shared_ptr 是为了解决 auto_ptr 在对象所有权上的局限性(auto_ptr 是独占的), 在使用引用计数的机制上提供了可以共享所有权的智能指针。

成员函数：

use_count 返回引用计数的个数

unique 返回是否是独占所有权(use_count 为 1)

swap 交换两个 shared_ptr 对象(即交换所拥有的对象)

reset 放弃内部对象的所有权或拥有对象的变更, 会引起原有对象的引用计数的减少

get 返回内部对象(指针), 由于已经重载了()方法, 因此和直接使用对象是一样的.如 shared_ptr sp(new int(1)); sp 与 sp.get()是等价的

4. weak_ptr

weak_ptr 是一种不控制对象生命周期的智能指针, 它指向一个 shared_ptr 管理的对象. 进行该对象的内存管理的是那个强引用的 shared_ptr. weak_ptr只是提供了对管理对象的一个访问手段。weak_ptr 设计的目的是为配合 shared_ptr 而引入的一种智能指针来协助 shared_ptr 工作, 它只可以从一个 shared_ptr 或另一个 weak_ptr 对象构造, 它的构造和析构不会引起引用记数的增加或减少。weak_ptr是用来解决shared_ptr相互引用时的死锁问题,如果说两个shared_ptr相互引用,那么这两个指针的引用计数永远不可能下降为0,资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和shared_ptr之间可以相互转化，shared_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared_ptr。

```
class B;
class A{
    public:
    shared_ptr<B> pb_;
    ~A() {
    cout<<"A delete\n";
    }
};
class B{
    public:
    shared_ptr<A> pa_;
    ~B(){
    cout<<"B delete\n";
    }
};
void fun(){
    shared_ptr<B> pb(new B());
    shared_ptr<A> pa(new A());
    pb->pa_ = pa;
    pa->pb_ = pb;
    cout<<pb.use_count()<<endl;
    cout<<pa.use_count()<<endl;
}
int main(){
    fun();
    return 0;
}
```

可以看到fun函数中pa ，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减一，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A B的析构函数没有被调用），如果把其中一个改为weak_ptr就可以了，我们把类A里面的shared_ptr pb*; 改为weak_ptr pb*; 运行结果如下，这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减一，同时pa析构时使A的计数减一，那么A的计数为0，A得到释放。

注意的是我们不能通过weak_ptr直接访问对象的方法，比如B对象中有一个方法print(),我们不能这样访问，pa->pb->*print(); 英文pb*是一个weak_ptr，应该先把它转化为shared_ptr,如：shared_ptr p = pa->pb_.lock(); p->print();

怎么判断一个数是二的倍数，怎么求一个数中有几个1，说一下你的思路并手写代码

1、判断一个数是不是二的倍数，即判断该数二进制末位是不是0：

a % 2 == 0 或者a & 0x0001 == 0。

2、求一个数中1的位数，可以直接逐位除十取余判断：

```
int fun(long x) {
    int _count = 0;
    while (x){
        if(x % 10 == 1)
        ++_count;
    x /= 10;
    }
    return _count;
}
int main() {
    cout << fun(123321) << endl;
    return 0;
}
```

请你回答一下野指针是什么？

指向一个已删除的对象或未申请访问受限内存区域的指针。

请回答一下数组和指针的区别

指针和数组的主要区别如下：

-指针	-数组
保存数据的地址	保存数据
间接访问数据，首先获得指针的内容，然后将其作为地址，从该地址中提取数据	直接访问数据，
通常用于动态的数据结构	通常用于固定数目且数据类型相同的元素
通过Malloc分配内存，free释放内存	隐式的分配和删除
通常指向匿名数据，操作匿名函数	自身即为数据名

请你介绍一下C++中的智能指针

智能指针主要用于管理在堆上分配的内存，它将普通的指针封装为一个栈对象。当栈对象的生存周期结束后，会在析构函数中释放掉申请的内存，从而防止内存泄漏。C++ 11中最常用的智能指针类型为shared_ptr,它采用引用计数的方法，记录当前内存资源被多少个智能指针引用。该引用计数的内存在堆上分配。当新增一个时引用计数加1，当过期时引用计数减一。只有引用计数为0时，智能指针才会自动释放引用的内存资源。对shared_ptr进行初始化时不能将一个普通指针直接赋值给智能指针，因为一个是指针，一个是类。可以通过make_shared函数或者通过构造函数传入普通指针。并可以通过get函数获得普通指针。

请你回答一下智能指针有没有内存泄露的情况

参考网址

循环引用问题可以参考这个链接上的问题理解，“循环引用”简单来说就是：两个对象互相使用一个shared_ptr成员变量指向对方的会造成循环引用。导致引用计数失效。下面给段代码来说明循环引用：

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

class B;
class A
{
public:// 为了省去一些步骤这里 数据成员也声明为public
    //weak_ptr<B> pb;
    shared_ptr<B> pb;
    void doSomthing()
    {
//        if(pb.lock())
//        {
//
//        }
    }

    ~A()
    {
        cout << "kill A\n";
    }
};

class B
{
public:
    //weak_ptr<A> pa;
    shared_ptr<A> pa;
    ~B()
    {
        cout <<"kill B\n";
    }
};
```

```
int main(int argc, char** argv)
{
    shared_ptr<A> sa(new A());
    shared_ptr<B> sb(new B());
    if(sa && sb)
    {
        sa->pb=sb;
        sb->pa=sa;
    }
    cout<<"sa use count:"<<sa.use_count()<<endl;
    return 0;
}
```

上面的代码运行结果为：sa use count:2， 注意此时sa,sb都没有释放，产生了内存泄露问题！！！ 即A内部有指向B，B内部有指向A，这样对于A，B必定是在A析构后B才析构，对于B，A必定是在B析构后才析构A，这就是循环引用问题，违反常规，导致内存泄露。

一般来讲，解除这种循环引用有下面有三种可行的方法(参考)：

1. 当只剩下最后一个引用的时候需要手动打破循环引用释放对象。
2. 当A的生存期超过B的生存期的时候，B改为使用一个普通指针指向A。
3. 使用弱引用的智能指针打破这种循环引用。 虽然这三种方法都可行，但方法1和方法2都需要程序员手动控制，麻烦且容易出错。我们一般使用第三种方法：弱引用的智能指针weak_ptr。

强引用和弱引用 一个强引用当被引用的对象活着的话，这个引用也存在（就是说，当至少有一个强引用，那么这个对象就不能被释放）。share_ptr就是强引用。相对而言，弱引用当引用的对象活着的时候不一定存在。仅仅是当它存在的时候的一个引用。弱引用并不修改该对象的引用计数，这意味这弱引用它并不对对象的内存进行管理，在功能上类似于普通指针，然而一个比较大的区别是，弱引用能检测到所管理的对象是否已经被释放，从而避免访问非法内存。

使用weak_ptr来打破循环引用

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

class B;
class A
{
public:// 为了省去一些步骤这里 数据成员也声明为public
    weak_ptr<B> pb;
    //shared_ptr<B> pb;
    void doSomthing()
    {
        shared_ptr<B> pp = pb.lock();
        if(pp)//通过lock()方法来判断它所管理的资源是否被释放
        {
            cout<<"sb use count:"<<pp.use_count()<<endl;
        }
    }

    ~A()
```

```
    {
        cout << "kill A\n";
    }
};

class B
{
public:
    //weak_ptr<A> pa;
    shared_ptr<A> pa;
    ~B()
    {
        cout <<"kill B\n";
    }
};

int main(int argc, char** argv)
{
    shared_ptr<A> sa(new A());
    shared_ptr<B> sb(new B());
    if(sa && sb)
    {
        sa->pb=sb;
        sb->pa=sa;
    }
    sa->doSomthing();
    cout<<"sb use count:"<<sb.use_count()<<endl;
    return 0;
}
```

weak_ptr除了对所管理对象的基本访问功能（通过get()函数）外，还有两个常用的功能函数：expired()用于检测所管理的对象是否已经释放；lock()用于获取所管理的对象的强引用指针。不能直接通过weak_ptr来访问资源。那么如何通过weak_ptr来间接访问资源呢？答案是：在需要访问资源的时候weak_ptr为你生成一个shared_ptr，shared_ptr能够保证在shared_ptr没有被释放之前，其所管理的资源是不会被释放的。创建shared_ptr的方法就是lock()方法。

请你理解的c++中的引用和指针

定义： 1、引用：

C++是C语言的继承，它可进行过程化程序设计，又可以进行以抽象数据类型为特点的基于对象的程序设计，还可以进行以继承和多态为特点的面向对象的程序设计。引用就是C++对C语言的重要扩充。引用就是某一变量的一个别名，对引用的操作与对变量直接操作完全一样。引用的声明方法：类型标识符 &引用名=目标变量名；引用引入了对象的一个同义词。定义引用的表示方法与定义指针相似，只是用&代替了*。

2、指针：

指针利用地址，它的值直接指向存在电脑存储器中另一个地方的值。由于通过地址能找到所需的变量单元，可以说，地址指向该变量单元。因此，将地址形象化的称为“指针”。意思是通过它能找到以它为地址的内存单元。

区别：

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、可以有const指针，但是没有const引用；

6、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；

7、指针可以有多级指针（**p），而引用至于一级；

8、指针和引用使用++运算符的意义不一样；

9、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。

请你回答一下为什么析构函数必须是虚函数？为什么C++默认的析构函数不是虚函数 考点:虚函数 析构函数

参考地址

析构函数不一定必须是虚函数，是否为虚函数取决于该类的。

一般该类为基类产生继承和多态时，才会是虚函数，单独使用可以不是虚函数。

之所以在继承和多态时设计为虚函数是因为当new派生类并且用基类指针指向这个派生类，在销毁基类指针时只会调用基类的析构函数，不会调用派生类的析构函数，因为基类无法操作派生类中非继承的成员，这样就造成派生类只new无法delete造成内存泄露。

默认不是虚析构函数是因为如果析构函数为虚函数就需要编译器在类中增加虚函数表来实现虚函数机制，这样所需内存空间就更大了，因此没有必要默认为虚析构函数。

将可能会被继承的父类的析构函数设置为虚函数，可以保证当我们new一个子类，然后使用基类指针指向该子类对象，释放基类指针时可以释放掉子类的空间，防止内存泄漏。

C++默认的析构函数不是虚函数是因为虚函数需要额外的虚函数表和虚表指针，占用额外的内存。而对于不会被继承的类来说，其析构函数如果是虚函数，就会浪费内存。因此C++默认的析构函数不是虚函数，而是只有当需要当作父类时，设置为虚函数。

请你来说一下函数指针

参考网址

1、定义 函数指针是指向函数的指针变量。

函数指针本身首先是一个指针变量，该指针变量指向一个具体的函数。这正如用指针变量可指向整型变量、字符型、数组一样，这里是指向函数。

C在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。有了指向函数的指针变量后，可用该指针变量调用函数，就如同用指针变量可引用其他类型变量一样，在这些概念上是大体一致的。

2、用途：

调用函数和做函数的参数，比如回调函数。

3、示例：

```
char * fun(char * p)  {…}       // 函数fun
char * (*pf)(char * p);             // 函数指针pf
pf = fun;                          // 函数指针pf指向函数fun
pf(p);                          // 通过函数指针pf调用函数fun
```

请你来说一下fork函数

Fork：创建一个和当前进程映像一样的进程可以通过fork( )系统调用：

```
#include <sys/types.h>

#include <unistd.h>

pid_t fork(void);
```

成功调用fork( )会创建一个新的进程，它几乎与调用fork( )的进程一模一样，这两个进程都会继续运行。在子进程中，成功的fork( )调用会返回0。在父进程中fork( )返回子进程的pid。如果出现错误，fork( )返回一个负值。

最常见的fork( )用法是创建一个新的进程，然后使用exec( )载入二进制映像，替换当前进程的映像。这种情况下，派生（fork）了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。

在早期的Unix系统中，创建进程比较原始。当调用fork时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页的复制到子进程的地址空间中。但从内核角度来说，逐页的复制方式是十分耗时的。现代的Unix系统采取了更多的优化，例如Linux，采用了写时复制的方法，而不是对父进程空间进程整体复制。

请你来说一下C++中析构函数的作用

析构函数与构造函数对应，当对象结束其生命周期，如对象所在的函数已调用完毕时，系统会自动执行析构函数。

析构函数名也应与类名相同，只是在函数名前面加一个位取反符~，例如~stud( )，以区别于构造函数。它不能带任何参数，也没有返回值（包括void类型）。只能有一个析构函数，不能重载。

如果用户没有编写析构函数，编译系统会自动生成一个缺省的析构函数（即使自定义了析构函数，编译器也总是会为我们合成一个析构函数，并且如果自定义了析构函数，编译器在执行时会先调用自定义的析构函数再调用合成的析构函数），它也不进行任何操作。所以许多简单的类中没有用显式的析构函数。

如果一个类中有指针，且在使用的过程中动态的申请了内存，那么最好显示构造析构函数在销毁类之前，释放掉申请的内存空间，避免内存泄漏。

类析构顺序：1）派生类本身的析构函数；2）对象成员析构函数；3）基类析构函数。

请你来说一下静态函数和虚函数的区别

静态函数在编译的时候就已经确定运行时机，虚函数在运行的时候动态绑定。虚函数因为用了虚函数表机制，调用的时候会增加一次内存开销

虚函数用来实现动态多态，父类函数加virtual关键字，子类同名函数（完全和父类一样的函数，只是方法体不同，也叫重写）可加可不加。如当父类指针指向子类对象时，调用的是子类的函数。

静态成员函数只能在该类中访问静态成员变量（普通变量不能访问），在本类中有效，其他类不能访问该函数，静态成员函数为类所有不为对象所有，对象调用时不传this指针。

普通成员函数能访问本类中所有类型控制的成员变量。 随着对象的销毁，会自动调用析构函数，如果类中定义了析构函数，就调用此析构函数，如果没有定义，调用系统自带的析构函数。

参考网址

1.virtual与静态函数

C++中，静态成员函数不能被声明为virtual函数。

```
#include<iostream>
class Test
{
   public:
      // 编译错误：static成员函数不能声明为virtual
      virtual static void fun()  { }
};
```

```
#include<iostream>
class Test
{
   public:
      // 编译错误: static成员函数不能为const
      static void fun() const { }

      // 如果声明为下面这样，是可以的。
      const static void fun() {}
      或类似于
      const static int fun() { return 0; }
};
```

2.为何static成员函数不能为virtual

1. static成员不属于任何类对象或类实例，所以即使给此函数加上virutal也是没有任何意义的。
2. 静态与非静态成员函数之间有一个主要的区别。那就是静态成员函数没有this指针。

虚函数依靠vptr和vtable来处理。vptr是一个指针，在类的构造函数中创建生成，并且只能用this指针来访问它，因为它是类的一个成员，并且vptr指向保存虚函数地址的vtable.

对于静态成员函数，它没有this指针，所以无法访问vptr. 这就是为何static函数不能为virtual.

虚函数的调用关系：this -> vptr -> vtable ->virtual function

通过下面例子可以确定，当类增加了一个虚函数后，类的大小会增大4字节(指针的大小).

```
class Test
{
public:
	int _m;
};

sizeof(Test) = 4;

加入虚函数后，

class Test
{
public:
	int _m;
	virtual void fun();
};

//sizeof(Test) = 8
```

3.为何static成员函数不能为const函数

当声明一个非静态成员函数为const时，对this指针会有影响。对于一个Test类中的const修饰的成员函数，this指针相当于Test const *, 而对于非const成员函数，this指针相当于Test *.

而static成员函数没有this指针，所以使用const来修饰static成员函数没有任何意义。

volatile的道理也是如此。

```

public:
	int _m;
	virtual void fun();
};

//sizeof(Test) = 8
```

4.static 可以继承吗?

参考网址

1. 父类的static变量和函数在派生类中依然可用，但是受访问性控制（比如，父类的private域中的就不可访问），而且对static变量来说，派生类和父类中的static变量是共用空间的，这点在利用static变量进行引用计数的时候要特别注意。

2. static函数没有“虚函数”一说。因为static函数实际上是“加上了访问控制的全局函数”，全局函数哪来的什么虚函数？
3. 派生类的friend函数可以访问派生类本身的一切变量，包括从父类继承下来的protected域中的变量。但是对父类来说，他并不是friend的。

继承可以使得子类具有父类的各种属性和方法，而不需要再次编写相同的代码。

请你来说一说重载，覆盖和重写的区别

Overload(重载)：在C++程序中，可以将语义、功能相似的几个函数用同一个名字表示，但参数或返回值不同（包括类型、顺序不同），即函数重载。

（1）相同的范围（在同一个类中）；（2）函数名字相同；（3）参数不同；（4）virtual 关键字可有可无。

Override(覆盖)：是指派生类函数覆盖基类函数，特征是：（1）不同的范围（分别位于派生类与基类）；（2）函数名字相同；（3）参数相同；（4）基类函数必须有virtual 关键字。

Overwrite(重写)：是指派生类的函数屏蔽了与其同名的基类函数，规则如下：（1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual 关键字，基类的函数将被隐藏（注意别与重载混淆）。（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

请你说一说strcpy和strlen

strcpy是字符串拷贝函数，原型：

char *strcpy(char* dest, const char *src);

从src逐字节拷贝到dest，直到遇到'\0'结束，因为没有指定长度，可能会导致拷贝越界，造成缓冲区溢出漏洞,安全版本是strncpy函数。 strlen函数是计算字符串长度的函数，返回从开始到'\0'之间的字符个数。

stacpy的实现

```
char * strcpy( char *strDest, const char *strSrc )
{
    assert( (strDest != NULL) && (strSrc != NULL) );
    char *address = strDest;
    while( (*strDest++ = *strSrc++) != ‘\0’ );
    return address;
 }
```

//无bug版本

```
char * Strcpy(char *strDest, const char *strSrc) {
    assert( (strDest != NULL) && (strSrc) );
    char tmpSrc[strlen(strSrc) + 1];
    int i = 0;
    while( (tmpSrc[i++] = *strSrc++) );
    char *address = strDest;
    i = 0;
    while( (*strDest++ = tmpSrc[i++]) );
    return address;
}
```

strlen的实现**

```
/* strlen: return length of s */
int strlen(char s[])
{
    int i;
    while (s[i] != '\0') ++i;
    return i;
}
```

请你说一说你理解的虚函数和多态

多态的实现主要分为静态多态和动态多态，静态多态主要是重载，在编译的时候就已经确定；动态多态是用虚函数机制实现的，在运行期间动态绑定。举个例子：一个父类类型的指针指向一个子类对象时候，使用父类的指针去调用子类中重写了的父类中的虚函数的时候，会调用子类重写过后的函数，在父类中声明为加了virtual关键字的函数，在子类中重写时候不需要加virtual也是虚函数。 虚函数的实现：在有虚函数的类中，类的最开始部分是一个虚函数表的指针，这个指针指向一个虚函数表，表中放了虚函数的地址，实际的虚函数在代码段(.text)中。当子类继承了父类的时候也会继承其虚函数表，当子类重写父类中虚函数时候，会将其继承到的虚函数表中的地址替换为重新写的函数地址。使用了虚函数，会增加访问内存开销，降低效率。

请你来回答一下++i和i++的区别

1. ++i称先自增，把储存器中i的值增加1，再取出来参与本次运算。
2. i++称后自增，先把储存器中i的值取出来参与本次运算，储存器中i的值再增加1。

++i的实现

```
int&  int::operator++ () {
         *this +=1;
          return *this;
     }
```

i++的实现

```
const int  int::operator (int) {
     int oldValue = *this;
     ++ (*this) ;
     return oldValue;
     }
```

请你来写个函数在main函数执行前先运行

```
__attribute__((constructor))
void func () {
    return ;
}
```

有段代码写成了下边这样，如果在只修改一个字符的前提下，使代码输出20个hello?

```
for(int i = 0; i < 20; i--)
cout << "hello" << endl;
```

参考答案

```
for(int i=0; i + 20; i--)
    cout << "hello" << endl;
```

请你来说一下智能指针shared_ptr的实现

> shared_ptr的一个最大的陷阱是循环引用，循环，循环引用会导致堆内存无法正确释放，导致内存泄漏。循环引用在weak_ptr中有介绍。
>
> **核心要理解引用计数，什么时候销毁底层指针，还有赋值，拷贝构造时候的引用计数的变化，析构的时候要判断底层指针的引用计数为0了才能真正释放底层指针的内存**

shared_ptr是最常用的智能指针（项目中我只用过shared_ptr）。shared_ptr采用了引用计数器，多个shared_ptr中的T *ptr指向同一个内存区域（同一个对象），并共同维护同一个引用计数器。shared_ptr定义如下,记录同一个实例被引用的次数，当引用次数大于0时可用，等于0时释放内存。

```
temple<typename T>
class SharedPtr {
public:
   ...
private:
    T *_ptr;
    int *_refCount;     //should be int*, rather than int
};
```

shared_ptr对象每次离开作用域时会自动调用析构函数，而析构函数并不像其他类的析构函数一样，而是在释放内存是先判断引用计数器是否为0。等于0才做delete操作，否则只对引用计数器左减一操作。

```
~SharedPtr()
{
    if (_ptr && --*_refCount == 0) {
        delete _ptr;
        delete _refCount;
    }
}
```

当用一个shared_ptr other去给另一个 shared_ptr sp赋值时，发生了两件事情：

1. sp指针指向发生变化，不再指向之前的内存区域，所以赋值前原来的_refCount要自减
2. sp指针指向other.ptr，所以other的引用计数器_refCount要做++操作。

```
SharedPtr &operator=(SharedPtr &other)
{
    if(this==&other)
        return *this;

    ++*other._refCount;
    if (--*_refCount == 0) {
        delete _ptr;
        delete _refCount;
    }

    _ptr = other._ptr;
    _refCount = other._refCount;
    return *this;
}
```

以下四行代码的区别是什么?

参考网址

```
const char * arr = "123";
char * brr = "123";
const char crr[] = "123";
char drr[] = "123";
```

const char * arr = "123";

//字符串123保存在常量区，const本来是修饰arr指向的值不能通过arr去修改，但是字符串“123”在常量区，本来就不能改变，所以加不加const效果都一样

char * brr = "123";

//字符串123保存在常量区，这个arr指针指向的是同一个位置，同样不能通过brr去修改"123"的值

const char crr[] = "123";

//这里123本来是在栈上的，但是编译器可能会做某些优化，将其放到常量区

char drr[] = "123";

//字符串123保存在栈区，可以通过drr去修改

请你来说一下C++里是怎么定义常量的？常量存放在内存的哪个位置

c++中定义常量有两种方法：

1. 使用`#define`预处理器：`#define MAX_N 1000`
2. 使用`const`关键字：`const int max = 1000;`
3. 因为无法解释const栈变量的存储位置。实际上并不存在常量存储区，只有全局/静态存储区。**const**类型的存储跟一般的变量没有区别，在外部定义的存储在全局数据区，static的存储在静态数据区，在函数内部定义的存储在栈，const跟非const存储上没区别，只不过是read only的。

常量在C++里的定义就是一个top-level const加上对象类型，常量定义必须初始化。对于局部对象，常量存放在栈区，对于全局对象，常量存放在全局/静态存储区。对于字面值常量，常量存放在常量存储区。

请你来回答一下const修饰成员函数的目的是什么

const修饰的成员函数表明函数调用不会对对象做出任何更改，事实上，如果确认不会对对象做更改，就应该为函数加上const限定，这样无论const对象还是普通对象都可以调用该函数。

如果同时定义了两个函数，一个带const，一个不带，会有问题吗？

不会，这相当于函数的重载。

在同一个类里，同时存在一个const方法和一个普通方法是没问题的。

请你来说一说隐式类型转换

首先，对于内置类型，低精度的变量给高精度变量赋值会发生隐式类型转换

其次，对于只存在单个参数的构造函数的对象构造来说，函数调用可以直接使用该参数传入，编译器会自动调用其构造函数生成临时对象。

- 为什么要进行隐式类型转换?

  C++面向对象的多态特性，就是通过父类的类型实现对子类的封装。

  通过隐式转换，你可以直接将一个子类的对象使用父类的类型进行返回。

  在比如，数值和布尔类型的转换，整数和浮点数的转换等。

  某些方面来说，隐式转换给C++程序开发者带来了不小的便捷。

  C++是一门强类型语言，类型的检查是非常严格的。

  如果没有类型的隐式转换，这将给程序开发者带来很多的不便。

  当然，凡事都有两面性，在你享受方便快捷的一面时，你不得不面对太过智能以至完全超出了你的控制。

  风险就在不知不觉间出现。、
- c++隐式类型转换的原则：
  - 基本数据类型 基本数据类型的转换以取值范围的作为转换基础（保证精度不丢失）。 隐式转换发生在从小->大的转换中。比如从char转换为int。 从int-》long。
  - 自定义对象 子类对象可以隐式的转换为父类对象。
- c++隐式转换发生的条件：

  > 低精度向高精度转换

  - 混合类型的算术运算表达式中：

    ```
    int a = 3;
    double = 4.5;
    a = b;
    ```

  - 不同类型的复制操作：

    ```
    int a = true;
    int * ptr = nullptr;
    ```

  - 函数参数传值：

    ```
    void func(double a);
    func(1);
    ```

  - 函数返回值：

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

说说你了解的类型转换

reinterpret_cast：可以用于任意类型的指针之间的转换，对转换的结果不做任何保证 dynamic_cast：这种其实也是不被推荐使用的，更多使用static_cast，dynamic本身只能用于存在虚函数的父子关系的强制类型转换，对于指针，转换失败则返回nullptr，对于引用，转换失败会抛出异常 const_cast：对于未定义const版本的成员函数，我们通常需要使用const_cast来去除const引用对象的const，完成函数调用。另外一种使用方式，结合static_cast，可以在非const版本的成员函数内添加const，调用完const版本的成员函数后，再使用const_cast去除const限定。
static_cast：完成基础数据类型；同一个继承体系中类型的转换；任意类型与空指针类型void* 之间的转换。

请你来说一说C++函数栈空间的最大值

默认是1M，不过可以调整

请你来说一说extern“C”

C++调用C函数需要extern C，因为C语言没有函数重载。

extern "C"的主要作用就是为了能够正确实现C++代码调用其他C语言代码。加上extern "C"后，会指示编译器这部分代码按C语言（而不是C++）的方式进行编译。由于C++支持函数重载，因此编译器编译函数的过程中会将函数的参数类型也加到编译后的代码中，而不仅仅是函数名；而C语言并不支持函数重载，因此编译C语言代码的函数时不会带上函数的参数类型，一般只包括函数名。

这个功能十分有用处，因为在C++出现以前，很多代码都是C语言写的，而且很底层的库也是C语言写的，为了更好的支持原来的C代码和已经写好的C语言库，需要在C++中尽可能的支持C，而extern "C"就是其中的一个策略。

这个功能主要用在下面的情况：

1. C++代码调用C语言代码
2. 在C++的头文件中使用
3. 在多个人协同开发时，可能有的人比较擅长C语言，而有的人擅长C++，这样的情况下也会有用到

为什么标准头文件都有类似的结构?

```
#ifndef __INCvxWorksh /*防止该头文件被重复引用*/
#define __INCvxWorksh
#ifdef __cplusplus              //告诉编译器，这部分代码按C语言的格式进行编译，而不是C++的
extern "C"{
#endif

/*…*/

#ifdef __cplusplus
}

#endif
#endif /*end of __INCvxWorksh*/
```

分析：

- 显然，头文件中编译宏"#ifndef **INCvxWorksh** 、**#define** INCvxWorksh、#endif"的作用是为了防止该头文件被重复引用

请你回答一下new/delete与malloc/free的区别是什么

首先，new/delete是C++的关键字，而malloc/free是C语言的库函数，

后者使用必须指明申请内存空间的大小，对于类类型的对象，后者不会调用构造函数和析构函数

new返回的是指定类型的指针，并且可以自动计算所申请内存的大小。而 malloc需要我们计算申请内存的大小，并且在返回时强行转换为实际类型的指针。

new 和 delete 到底是什么?

如果找工作的同学看一些面试的书，我相信都会遇到这样的题：sizeof 不是函数，然后举出一堆的理由来证明 sizeof 不是函数。在这里，和 sizeof 类似，new 和 delete 也不是函数，它们都是 C++ 定义的关键字，通过特定的语法可以组成表达式。和 sizeof 不同的是，sizeof 在编译时候就可以确定其返回值，new 和 delete 背后的机制则比较复杂。

继续往下之前，请你想想你认为 new 应该要做些什么？也许你第一反应是，new 不就和 C 语言中的 malloc 函数一样嘛，就用来动态申请空间的。你答对了一半，看看下面语句：

```
string *ps = new string("hello world");
```

你就可以看出 new 和 malloc 还是有点不同的，malloc 申请完空间之后不会对内存进行必要的初始化，而 new 可以。所以 new expression 背后要做的事情不是你想象的那么简单。在我用实例来解释 new 背后的机制之前，你需要知道 operator new 和 operator delete 是什么玩意。

operator new 和 operator delete

这两个其实是 C++ 语言标准库的库函数，原型分别如下：

```
void *operator new(size_t);     //allocate an object
void *operator delete(void *);    //free an object

void *operator new[](size_t);     //allocate an array
void *operator delete[](void *);    //free an array
```

后面两个你可以先不看，后面再介绍。前面两个均是 C++ 标准库函数，你可能会觉得这是函数吗？请不要怀疑，这就是函数！C++ Primer 一书上说这不是重载 new 和 delete 表达式（如 operator= 就是重载 = 操作符），因为 new 和 delete 是不允许重载的。但我还没搞清楚为什么要用 operator new 和 operator delete 来命名，比较费解。我们只要知道它们的意思就可以了，这两个函数和 C 语言中的 malloc 和 free 函数有点像了，都是用来申请和释放内存的，并且 operator new 申请内存之后不对内存进行初始化，直接返回申请内存的指针。

我们可以直接在我们的程序中使用这几个函数。

new 和 delete 背后机制

知道上面两个函数之后，我们用一个实例来解释 new 和 delete 背后的机制：

我们不用简单的 C++ 内置类型来举例，使用复杂一点的类类型，定义一个类 A：

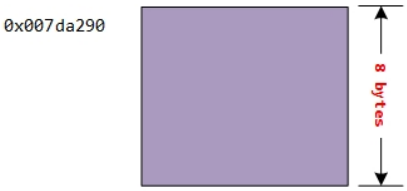
```
class A
{
public:
    A(int v) : var(v)
    {
        fopen_s(&file, "test", "r");
    }
    ~A()
    {
        fclose(file);
    }

private:
    int var;
    FILE *file;
};
```

很简单，类 A 中有两个私有成员，有一个构造函数和一个析构函数，构造函数中初始化私有变量 var 以及打开一个文件，析构函数关闭打开的文件。

我们使用

```
class *pA = new A(10);
```

来创建一个类的对象，返回其指针 pA。如下图所示 new 背后完成的工作：

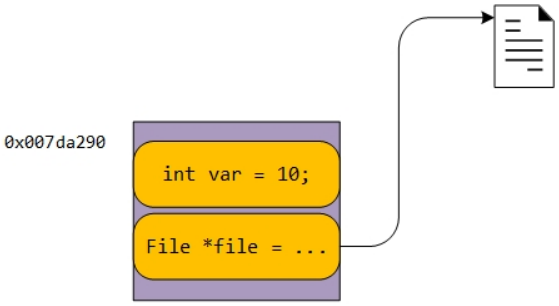


① 调用 **operator new** 标准库函数，分配足够大的原始的未类型化的内存

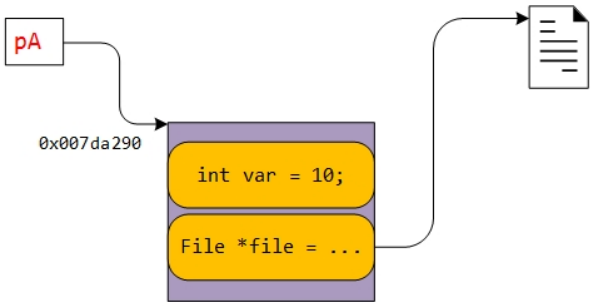
当使用 **new** 表达式的时候，如

class A *pA = new A(10);

实际上发生的事情如右图所示。



② 运行该类型的相应的构造函数，在上一步分配的内存上进行初始化对象



③ 返回指向新分配并构造的对象的指针

类 A 的定义：

```
class A
{
public:
    A(int v) : var(v)
    {
        fopen_s(&file, "test", "r");
    }
    ~A()
    {
        fclose(file);
    }

private:
    int var;
    FILE *file;
};
```

简单总结一下：

首先需要调用上面提到的 operator new 标准库函数，传入的参数为 class A 的大小，这里为 8 个字节，至于为什么是 8 个字节，你可以看看《深入 C++ 对象模型》一书，这里不做多解释。这样函数返回的是分配内存的起始地址，这里假设是 0x007da290。 上面分配的内存是未初始化的，也是未类型化的，第二步就在这一块原始的内存上对类对象进行初始化，调用的是相应的构造函数，这里是调用 A:A(10); 这个函数，从图中也可以看到对这块申请的内存进行了初始化，var=10, file 指向打开的文件。 最后一步就是返回新分配并构造好的对象的指针，这里 pA 就指向 0x007da290 这块内存，pA 的类型为类 A 对象的指针。

所有这三步，你都可以通过反汇编找到相应的汇编代码，在这里我就不列出了.

好了，那么 delete 都干了什么呢？还是接着上面的例子，如果这时想释放掉申请的类的对象怎么办？当然我们可以使用下面的语句来完成：

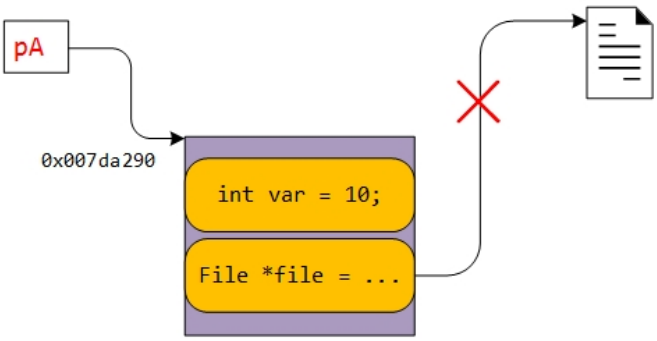
```
delete pA;
```

delete 所做的事情如下图所示：

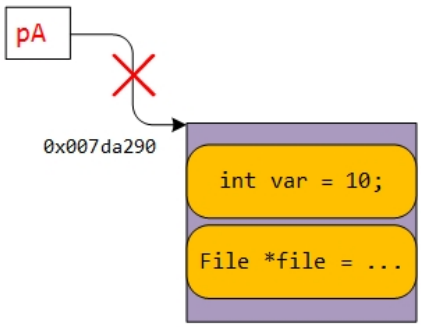
当使用 **delete** 表达式的时候，如

delete pA;

实际上发生的事情如右下图所示。



① 对 **pA** 指向的对象运行析构函数



② 调用 **operator delete** 标准库函数释放该对象所用的内存

delete 就做了两件事情：

调用 pA 指向对象的析构函数，对打开的文件进行关闭。 通过上面提到的标准库函数 operator delete 来释放该对象的内存，传入函数的参数为 pA 的值，也就是 0x007d290。

好了，解释完了 new 和 delete 背后所做的事情了，是不是觉得也很简单？不就多了一个构造函数和析构函数的调用嘛。

如何申请和释放一个数组？ 我们经常要用到动态分配一个数组，也许是这样的：

```
string *psa = new string[10];      //array of 10 empty strings
int *pia = new int[10];           //array of 10 uninitialized ints
```

上面在申请一个数组时都用到了 new [] 这个表达式来完成，按照我们上面讲到的 new 和 delete 知识，第一个数组是 string 类型，分配了保存对象的内存空间之后，将调用 string 类型的默认构造函数依次初始化数组中每个元素；第二个是申请具有内置类型的数组，分配了存储 10 个 int 对象的内存空间，但并没有初始化。

如果我们想释放空间了，可以用下面两条语句：

```
delete [] psa;
delete [] pia;
```

都用到 delete [] 表达式，注意这地方的 [] 一般情况下不能漏掉！我们也可以想象这两个语句分别干了什么：第一个对 10 个 string 对象分别调用析构函数，然后再释放掉为对象分配的所有内存空间；第二个因为是内置类型不存在析构函数，直接释放为 10 个 int 型分配的所有内存空间。

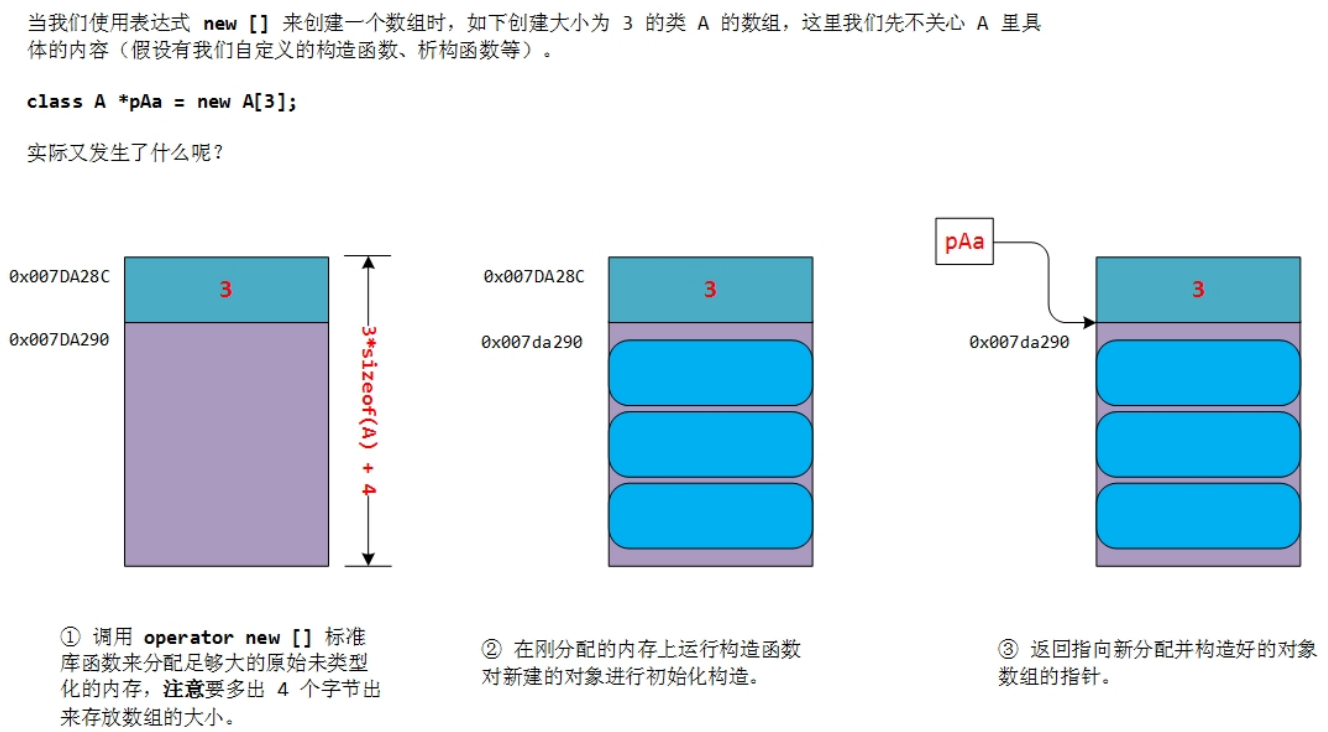
这里对于第一种情况就有一个问题了：我们如何知道 psa 指向对象的数组的大小？怎么知道调用几次析构函数？

这个问题直接导致我们需要在 new [] 一个对象数组时，需要保存数组的维度，C++ 的做法是在分配数组空间时多分配了 4 个字节的大小，专门保存数组的大小，在 delete [] 时就可以取出这个保存的数，就知道了需要调用析构函数多少次了。

还是用图来说明比较清楚，我们定义了一个类 A，但不具体描述类的内容，这个类中有显示的构造函数、析构函数等。那么 当我们调用

```
class A *pAa = new A[3];
```

时需要做的事情如下：



从这个图中我们可以看到申请时在数组对象的上面还多分配了 4 个字节用来保存数组的大小，但是最终返回的是对象数组的指针，而不是所有分配空间的起始地址。

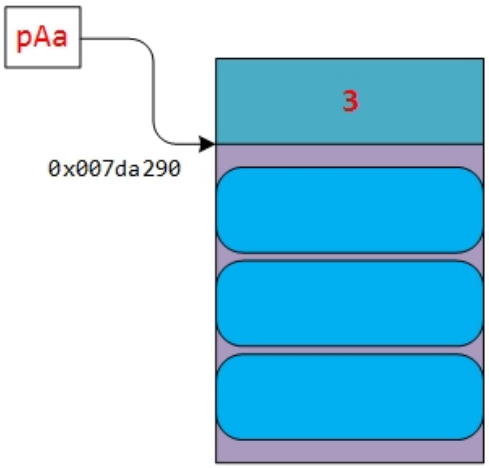
这样的话，释放就很简单了：

```
delete [] pAa;
```

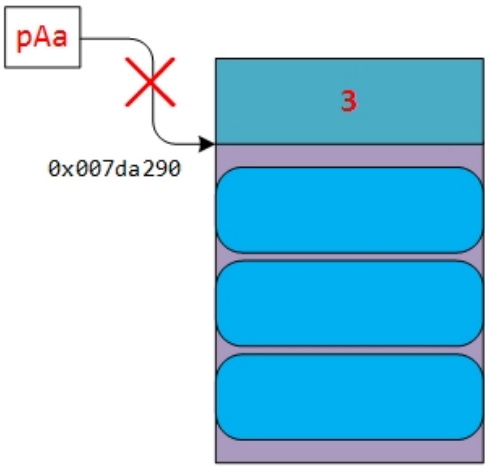
当我们使用表达式 **delete []** 来释放申请的数组，如下

delete pAa;

实际又发生了什么呢？



① 对数组中各个对象运行析构函数，数组的维数保存在 pA 前 4 个字节里。（**注：**析构函数不能明显的反应，故没画出）



② 调用 **operator delete []** 标准库函数释放申请的空间。（**注意：**不仅仅释放对象数组所占的空间，还有上面的 4 个字节！）

这里要注意的两点是：

- 调用析构函数的次数是从数组对象指针前面的 4 个字节中取出；
- 传入 `operator delete[]` 函数的参数不是数组对象的指针 pAa，而是 pAa 的值减 4。

为什么 new/delete 、new []/delete[] 要配对使用？ 其实说了这么多，还没到我写这篇文章的最原始意图。从上面解释的你应该懂了 new/delete、new[]/delete[] 的工作原理了，因为它们之间有差别，所以需要配对使用。但偏偏问题不是这么简单，这也是我遇到的问题，如下这段代码：

```
int *pia = new int[10];
delete []pia;
```

这肯定是没问题的，但如果把 delete []pia; 换成 delete pia; 的话，会出问题吗？

这就涉及到上面一节没提到的问题了。上面我提到了在 new [] 时多分配 4 个字节的缘由，因为析构时需要知道数组的大小，但如果不调用析构函数呢（如内置类型，这里的 int 数组）？我们在 new [] 时就没必要多分配那 4 个字节， delete [] 时直接到第二步释放为 int 数组分配的空间。如果这里使用 delete pia;那么将会调用 operator delete 函数，传入的参数是分配给数组的起始地址，所做的事情就是释放掉这块内存空间。不存在问题的。

这里说的使用 new [] 用 delete 来释放对象的提前是：对象的类型是内置类型或者是无自定义的析构函数的类类型！

我们看看如果是带有自定义析构函数的类类型，用 new [] 来创建类对象数组，而用 delete 来释放会发生什么？用上面的例子来说明：

```
class A *pAa = new class A[3];
delete pAa;
```

那么 delete pAa; 做了两件事：

调用一次 pAa 指向的对象的析构函数； 调用 operator delete(pAa); 释放内存。 显然，这里只对数组的第一个类对象调用了析构函数，后面的两个对象均没调用析构函数，如果类对象中申请了大量的内存需要在析构函数中释放，而你却在销毁数组对象时少调用了析构函数，这会造成内存泄漏。

上面的问题你如果说没关系的话，那么第二点就是致命的了！直接释放 pAa 指向的内存空间，这个总是会造成严重的段错误，程序必然会奔溃！因为分配的空间的起始地址是 pAa 指向的地方减去 4 个字节的地方。你应该传入参数设为那个地址！

同理，你可以分析如果使用 new 来分配，用 delete [] 来释放会出现什么问题？是不是总会导致程序错误？

总的来说，记住一点即可：new/delete、new[]/delete[] 要配套使用总是没错的！

请你说说你了解的RTTI

运行时类型检查，在C++层面主要体现在dynamic_cast和typeid,VS中虚函数表的-1位置存放了指向type_info的指针。对于存在虚函数的类型，typeid和dynamic_cast都会去查询type_info

详细网址

RTTI是"Runtime Type Information"的缩写，意思是运行时类型信息，它**提供了运行时确定对象类型的方法**。RTTI并不是什么新的东西，很早就有了这个技术，但是，在实际应用中使用的比较少而已。而我这里就是对RTTI进行总结，今天我没有用到，并不代表这个东西没用。学无止境，先从typeid函数开始讲起。

typeid函数

typeid的主要作用就是让用户知道当前的变量是什么类型的

```
#include <typeinfo>
float f = 1.0f;
double d = 2;
cout<<typeid(f).name()<<endl; // float
cout<<typeid(d).name()<<endl; // double
//结构体依然支持
```

是的，对于我们自定义的结构体和类，tpyeid都能支持。在上面的代码中，在调用完typeid之后，都会接着调用name()函数，可以看出typeid函数返回的是一个结构体或者类，然后，再调用这个返回的结构体或类的name成员函数；其实，typeid是一个返回类型为type_info类型的函数。那么，我们就有必要对这个type_info类进行总结一下，毕竟它实际上存放着类型信息。

type_info类

去掉那些该死的宏，在Visual Studio 2012中查看type_info类的定义如下：

```
```

```
class type_info
{
public:
    virtual ~type_info();
    bool operator==(const type_info& _Rhs) const; // 用于比较两个对象的类型是否相等
    bool operator!=(const type_info& _Rhs) const; // 用于比较两个对象的类型是否不相等
    bool before(const type_info& _Rhs) const;

    // 返回对象的类型名字，这个函数用的很多
    const char* name(__type_info_node* __ptype_info_node = &__type_info_root_node) const;
    const char* raw_name() const;
private:
    void *_M_data;
    char _M_d_name[1];
    type_info(const type_info& _Rhs);
    type_info& operator=(const type_info& _Rhs);
    static const char * _Name_base(const type_info *,__type_info_node* __ptype_info_node);
    static void _Type_info_dtor(type_info *);
};
```

在type_info类中，复制构造函数和赋值运算符都是私有的，同时也没有默认的构造函数；所以，我们没有办法创建type_info类的变量，例如type_info A;这样是错误的。那么typeid函数是如何返回一个type_info类的对象的引用的呢？我在这里不进行讨论，思路就是类的友元函数。

typeid函数的使用

typeid使用起来是非常简单的，常用的方式有以下两种：

1.使用type_info类中的name()函数返回对象的类型名称

就像上面的代码中使用的那样；但是，这里有一点需要注意，比如有以下代码：

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

class A
{
public:
     void Print() { cout<<"This is class A."<<endl; }
};

class B : public A
{
public:
     void Print() { cout<<"This is class B."<<endl; }
};

int main()
{
     A *pA = new B();
     cout<<typeid(pA).name()<<endl; // class A *
     cout<<typeid(*pA).name()<<endl; // class A
     return 0;
```

```
}
```

我使用了两次typeid，但是两次的参数是不一样的；输出结果也是不一样的；当我指定为pA时，由于pA是一个A类型的指针，所以输出就为class A *；当我指定*pA时，它表示的是pA所指向的对象的类型，所以输出的是class A；所以需要区分typeid(*pA)和typeid(pA)的区别，它们两个不是同一个东西；但是，这里又有问题了，明明pA实际指向的是B，为什么得到的却是class A呢？我们在看下一段代码：

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

class A
{
public:
     virtual void Print() { cout<<"This is class A."<<endl; }
};

class B : public A
{
public:
     void Print() { cout<<"This is class B."<<endl; }
};

int main()
{
     A *pA = new B();
     cout<<typeid(pA).name()<<endl; // class A *
     cout<<typeid(*pA).name()<<endl; // class B
     return 0;
}
```

好了，我将Print函数变成了虚函数，输出结果就不一样了，这说明什么？这就是RTTI在捣鬼了，当类中不存在虚函数时，typeid是编译时期的事情，也就是静态类型，就如上面的cout<<typeid(*pA).name()<<endl;输出class A一样；当类中存在虚函数时，typeid是运行时期的事情，也就是动态类型，就如上面的cout<<typeid(*pA).name()<<endl;输出class B一样，关于这一点，我们在实际编程中，经常会出错，一定要谨记。

2.使用type_info类中重载的==和!=比较两个对象的类型是否相等

这个会经常用到，通常用于比较两个带有虚函数的类的对象是否相等，例如以下代码：

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

class A
{
public:
     virtual void Print() { cout<<"This is class A."<<endl; }
};

class B : public A
{
```

```
public:
     void Print() { cout<<"This is class B."<<endl; }
};

class C : public A
{
public:
     void Print() { cout<<"This is class C."<<endl; }
};

void Handle(A *a)
{
     if (typeid(*a) == typeid(A))
     {
          cout<<"I am a A truly."<<endl;
     }
     else if (typeid(*a) == typeid(B))
     {
          cout<<"I am a B truly."<<endl;
     }
     else if (typeid(*a) == typeid(C))
     {
          cout<<"I am a C truly."<<endl;
     }
     else
     {
          cout<<"I am alone."<<endl;
     }
}

int main()
{
     A *pA = new B();
     Handle(pA);
     delete pA;
     pA = new C();
     Handle(pA);
     return 0;
}
```

这是一种用法，呆会我再总结如何使用dynamic_cast来实现同样的功能。

dynamic_cast的内幕

在这篇《static_cast、dynamic_cast、const_cast和reinterpret_cast总结》的文章中，也介绍了dynamic_cast的使用，对于dynamic_cast到底是如何实现的，并没有进行说明，而这里就要对于dynamic_cast的内幕一探究竟。首先来看一段代码：

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

class A
```

```
{
public:
     virtual void Print() { cout<<"This is class A."<<endl; }
};

class B
{
public:
     virtual void Print() { cout<<"This is class B."<<endl; }
};

class C : public A, public B
{
public:
     void Print() { cout<<"This is class C."<<endl; }
};

int main()
{
     A *pA = new C;
     //C *pC = pA; // Wrong
     C *pC = dynamic_cast<C *>(pA);
     if (pC != NULL)
     {
          pC->Print();
     }
     delete pA;
}
```

在上面代码中，如果我们直接将pA赋值给pC，这样编译器就会提示错误，而当我们加上了dynamic_cast之后，一切就ok了。那么dynamic_cast在后面干了什么呢?

dynamic_cast主要用于在多态的时候，它允许在运行时刻进行类型转换，从而使程序能够在一个类层次结构中安全地转换类型，把基类指针（引用）转换为派生类指针（引用）。我在《COM编程——接口的背后》这篇博文中总结的那样，当类中存在虚函数时，编译器就会在类的成员变量中添加一个指向虚函数表的vptr指针，每一个class所关联的type_info object也经由virtual table被指出来，通常这个type_info object放在表格的第一个slot。当我们进行dynamic_cast时，编译器会帮我们进行语法检查。如果指针的静态类型和目标类型相同，那么就什么事情都不做；否则，首先对指针进行调整，使得它指向vftable，并将其和调整之后的指针、调整的偏移量、静态类型以及目标类型传递给内部函数。其中最后一个参数指明转换的是指针还是引用。两者唯一的区别是，如果转换失败，前者返回NULL，后者抛出bad_cast异常。对于在typeid函数的使用中所示例的程序，我使用dynamic_cast进行更改，代码如下:

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

class A
{
public:
     virtual void Print() { cout<<"This is class A."<<endl; }
};
```

```
class B : public A
{
public:
     void Print() { cout<<"This is class B."<<endl; }
};

class C : public A
{
public:
     void Print() { cout<<"This is class C."<<endl; }
};

void Handle(A *a)
{
     if (dynamic_cast<B*>(a))
     {
          cout<<"I am a B truly."<<endl;
     }
     else if (dynamic_cast<C*>(a))
     {
          cout<<"I am a C truly."<<endl;
     }
     else
     {
          cout<<"I am alone."<<endl;
     }
}

int main()
{
     A *pA = new B();
     Handle(pA);
     delete pA;
     pA = new C();
     Handle(pA);
     return 0;
}
```

这个是使用dynamic_cast进行改写的版本。实际项目中，这种方法会使用的更多点。

总结

我在这里总结了RTTI的相关知识，希望大家看懂了。这篇博文有点长，希望大家也耐心的看。总结了就会有收获。

请你说说虚函数表具体是怎样实现运行时多态的?

子类若重写父类虚函数，虚函数表中，该函数的地址会被替换，对于存在虚函数的类的对象，在VS中，对象的对象模型的头部存放指向虚函数表的指针，通过该机制实现多态。

参考网址

（1）每一个含有虚函数的类，都会生成虚表(virtual table)。这个表，记录了对象的动态类型，决定了执行此对象的虚成员函数的时候，真正执行的那一个成员函数。

（2）对于有多个基类的类对象，会有多个虚表，每一个基类对应一个虚表，同时，虚表的顺序和继承时的顺序相同。

（3）在每一个类对象所占用的内存中，虚指针位于最前边，每个虚指针指向对应的虚表。

请你说说C语言是怎么进行函数调用的?

每一个函数调用都会分配函数栈，在栈内进行函数执行过程。调用前，先把返回地址压栈，然后把当前函数的esp指针压栈。

请你说说C语言参数压栈顺序?

从右到左

参考网址

详细参考网址

程序栈底为高地址，栈顶为低地址，因此上面的实例可以说明函数参数入栈顺序的确是从右至左的。可到底为什么呢？查了一直些文献得知，参数入栈顺序是和具体编译器实现相关的。比如，Pascal语言中参数就是从左到右入栈的，有些语言中还可以通过修饰符进行指定，如Visual C++。即然两种方式都可以，为什么C语言要选择从右至左呢?

进一步发现，Pascal语言不支持可变长参数，而C语言支持这种特色，正是这个原因使得C语言函数参数入栈顺序为从右至左。具体原因为：C方式参数入栈顺序（从右至左）的好处就是可以**动态变化参数个数**。通过栈堆分析可知，自左向右的入栈方式，最前面的参数被压在栈底。除非知道参数个数，否则是无法通过栈指针的相对位移求得最左边的参数。这样就变成了左边参数的个数不确定，正好和动态参数个数的方向相反。

因此，C语言函数参数采用自右向左的入栈顺序，主要原因是为了支持可变长参数形式，C语言中可变参数都是从左到右，所以不管你有多少个参数反正将最右面的那个压入栈底，最左面的参数出入栈顶。换句话说，如果不支持这个特色，C语言完全和Pascal一样，采用自左向右的参数入栈方式。

请你说说C++如何处理返回值

生成一个临时变量，把它的引用作为函数参数传入函数内。

请你回答一下C++中拷贝赋值函数的形参能否进行值传递?

不能。如果是这种情况下，调用拷贝构造函数的时候，首先要将实参传递给形参，这个传递的时候又要调用拷贝构造函数。。如此循环，无法完成拷贝，栈也会满。

无限递归。

```
#include <iostream.h>

class CExample
{
    int m_nTest;
public:

    CExample(int x):m_nTest(x) //带参数构造函数
    {
       cout << "constructor with argument/n";
    }

    CExample(const CExample & ex) //拷贝构造函数
    {
        m_nTest = ex.m_nTest;
        cout << "copy constructor/n";
    }

    CExample& operator = (const CExample &ex)//赋值函数(赋值运算符重载)
    {
        cout << "assignment operator/n";
        m_nTest = ex.m_nTest;
        return *this;
    }

    void myTestFunc(CExample ex)
    {
    }
};

int main()
{
    CExample aaa(2);
    CExample bbb(3);
    bbb = aaa;
    CExample ccc = aaa;
    bbb.myTestFunc(aaa);

    return 0;
}
```

看这个例子的输出结果：

```
constructor with argument     //CExample aaa(2);
constructor with argument     //CExample bbb(3);
assignment operator           //bbb = aaa;
copy constructor              //CExample ccc = aaa;
copy constructor              //  bbb.myTestFunc(aaa);
```

构造ccc，实质上是ccc.CExample(aaa); 我们假如拷贝构造函数参数不是引用类型的话， 那么将使得ccc.CExample(aaa)变成aaa传值给ccc.CExample(CExample ex)，即CExample ex = aaa，因为 ex 没有被初始化，所以 CExample ex = aaa 继续调用拷贝构造函数，接下来的是构造ex，也就是 ex.CExample(aaa)，必然又会有aaa传给CExample(CExample ex), 即 CExample ex = aaa;那么又会触发拷贝构造函数，就这下永远的递归下去。

请你回答一下malloc与new区别

malloc需要给定申请内存的大小，返回的指针需要强转。 new会调用构造函数，不用指定内存大小，返回的指针不用强转。

请你说一说select

select在使用前，先将需要监控的描述符对应的bit位置1，然后将其传给select,当有任何一个事件发生时，select将会返回所有的描述符，需要在应用程序自己遍历去检查哪个描述符上有事件发生，效率很低，并且其不断在内核态和用户态进行描述符的拷贝，开销很大

selcet函数是一个轮循函数，即当循环询问文件节点，可设置超时时间，超时时间到了就跳过代码继续往下执行。select()的机制中提供一fd_set的数据结构，实际上是一long类型的数组， 每一个数组元素都能与一打开的文件句柄(不管是Socket句柄，还是其他 文件或命名管道或设备句柄)建立联系，建立联系的工作由程序员完成， 当调用select()时，由内核根据IO状态修改fd_set的内容，由此来通知执 行了select()的进程哪一Socket或文件可读或可写。主要用于Socket通信当中。

请你说说fork,wait,exec函数

父进程产生子进程使用fork拷贝出来一个父进程的副本，此时只拷贝了父进程的页表，两个进程都读同一块内存，当有进程写的时候使用写实拷贝机制分配内存，exec函数可以加载一个elf文件去替换父进程，从此父进程和子进程就可以运行不同的程序了。fork从父进程返回子进程的pid，从子进程返回0.调用了wait的父进程将会发生阻塞，直到有子进程状态改变,执行成功返回0，错误返回-1。exec执行成功则子进程从新的程序开始运行，无返回值，执行失败返回-1

请你回答一下静态函数和虚函数的区别

静态函数在编译的时候就已经确定运行时机，虚函数在运行的时候动态绑定。虚函数因为用了虚函数表机制，调用的时候会增加一次内存开销

逻辑角度说静态员函数绑定class旦定义应class改变；虚函数意义恰恰运行态选择调用哪class同名函数本质矛盾同使用实现角度说静态成员函数本质与C函数致应内存固定址；虚函数应虚表索引值运行通索引值进行间接寻址两者兼容所论逻辑合理性实现行性说虚函数都必须非静态员函数