1、C++的多态如何实现

C++的多态性，**一言以蔽之**就是：

在基类的函数前加上 **virtual** 关键字，在派生类中重写该函数，运行时将会根据所指对象的实际类型来调用相应的函数，如果对象类型是派生类，就调用派生类的函数，如果对象类型是基类，就调用基类的函数。

举个例子：

```
#include <iostream>
using namespace std;
class Base{
public:
        virtual void fun(){
                cout << " Base::func()" <<endl;
        }
};
class Son1 : public Base{
public:
        virtual void fun() override{
                cout << " Son1::func()" <<endl;
        }
};
class Son2 : public Base{

};
int main(){
        Base* base = new Son1;
        base->fun();
        base = new Son2;
        base->fun();
        delete base;
        base = NULL;
        return 0;
}// 运行结果
// Son1::func()
// Base::func()
```

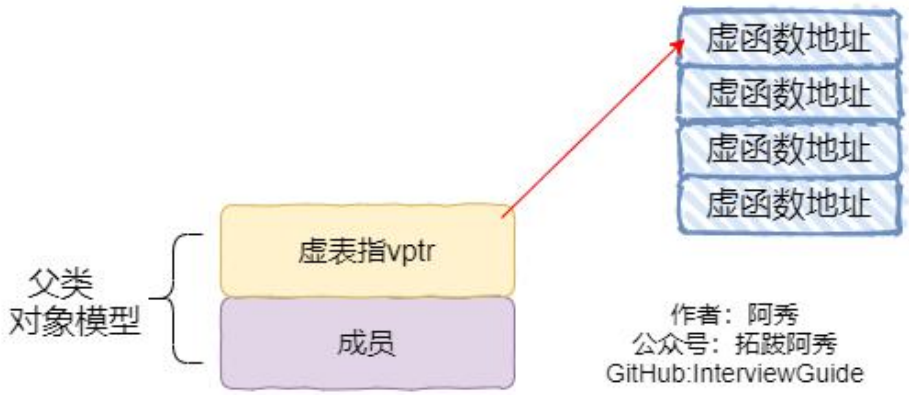
例子中，Base 为基类，其中的函数为虚函数。

子类 1 继承并重写了基类的函数，子类 2 继承基类但没有重写基类的函数，从结果分析子类体现了多态性，那么为什么会出现多态性，其底层的原理是什么？

这里需要引出虚表和虚基表指针的概念。

虚表：虚函数表的缩写，类中含有 virtual 关键字修饰的方法时，编译器会自动生成虚表

虚表指针：在含有虚函数的类实例化对象时，对象地址的前四个字节存储的指向虚表的指针



上图中展示了虚表和虚表指针在基类对象和派生类对象中的模型，下面阐述实现多态的过程：

（1）编译器在发现基类中有虚函数时，会自动为每个含有虚函数的类生成一份虚表，该表是一个一维数组，虚表里保存了虚函数的入口地址

（2）编译器会在每个对象的前四个字节中保存一个虚表指针，即 vptr，指向对象所属类的虚表。在构造时，根据对象的类型去初始化虚指针 vptr，从而让 vptr 指向正确的虚表，从而在调用虚函数时，能找到正确的函数

（3）所谓的合适时机，在派生类定义对象时，程序运行会自动调用构造函数，在构造函数中创建虚表并对虚表初始化。

在构造子类对象时，会先调用父类的构造函数。此时，编译器只"看到了"父类，并为父类对象初始化虚表指针，令它指向父类的虚表；当调用子类的构造函数时，为子类对象初始化虚表指针，令它指向子类的虚表

（4）当派生类对基类的虚函数没有重写时，派生类的虚表指针指向的是基类的虚表；当派生类对基类的虚函数重写时，派生类的虚表指针指向的是自身的虚表；当派生类中有自己的虚函数时，在自己的虚表中将此虚函数地址添加在后面

这样指向派生类的基类指针在运行时，就可以根据派生类对虚函数重写情况动态的进行调用，从而实现多态性。

2、为什么析构函数一般写成虚函数

由于类的多态性，基类指针可以指向派生类的对象，如果删除该基类的指针，就会调用该指针指向的派生类析构函数，而派生类的析构函数又自动调用基类的析构函数，这样整个派生类的对象完全被释放。

如果析构函数不被声明成虚函数，则编译器实施静态绑定，在删除基类指针时，只会调用基类的析构函数而不调用派生类析构函数，这样就会造成派生类对象析构不完全，造成内存泄漏。

所以将析构函数声明为虚函数是十分必要的。在实现多态时，当用基类操作派生类，在析构时防止只析构基类而不析构派生类的状况发生，要将基类的析构函数声明为虚函数。

```
#include <iostream>
using namespace std;
class Parent{
public:
        Parent(){
                cout << "Parent construct function"  << endl;
        };
        ~Parent(){
                cout << "Parent destructor function" <<endl;
        }
};
class Son : public Parent{
public:
        Son(){
                cout << "Son construct function"  << endl;
        };
        ~Son(){
                cout << "Son destructor function" <<endl;
        }
};
int main(){
        Parent* p = new Son();
        delete p;
        p = NULL;
        return 0;
}
//运行结果:
//Parent construct function
//Son construct function
//Parent destructor function
```

将基类的析构函数声明为虚函数：

```
#include <iostream>
using namespace std;
class Parent{
public:
        Parent(){
                cout << "Parent construct function"  << endl;
        };
        virtual ~Parent(){
                cout << "Parent destructor function" <<endl;
        }
};
class Son : public Parent{public:
        Son(){
                cout << "Son construct function"  << endl;
        };
        ~Son(){
                cout << "Son destructor function" <<endl;
        }
};
int main(){
        Parent* p = new Son();
        delete p;
        p = NULL;
        return 0;
}
//运行结果:
//Parent construct function
//Son construct function
//Son destructor function
//Parent destructor function
```

但存在一种特例，在 CRTP 模板中，不应该将析构函数声明为虚函数，理论上所有的父类函数都不应 该声明为虚函数，因为这种继承方式，不需要虚函数表。

3、构造函数能否声明为虚函数或者纯虚函数，析构函数呢？

析构函数：

- 析构函数可以为虚函数，并且一般情况下基类析构函数要定义为虚函数。
- 只有在基类析构函数定义为虚函数时，调用操作符 delete 销毁指向对象的基类指针时，才能准确调用派生类的析构函数（从该级向上按序调用虚函数），才能准确销毁数据。
- **析构函数可以是纯虚函数**，含有纯虚函数的类是抽象类，此时不能被实例化。但派生类中可以根据自身需求重新改写基类中的纯虚函数。

构造函数：

- 根据《effective C++》的条款 09：绝不在构造和析构过程中调用虚函数可知，在构造函数中虽然可以调用虚函数，但是强烈建议不要这样做。因为基类的构造的过程中，虚函数不能算作是虚函数。若构造函数中调用虚函数，可能会导致不确定行为的发生.
- 虚函数对应一个 vtable(虚函数表)，类中存储一个 vptr 指向这个 vtable。如果构造函数是虚函数，就需要通过 vtable 调用，可是对象没有初始化就没有 vptr，无法找到 vtable，所以构造函数不能是虚函数。

4、基类的虚函数表存放在内存的什么区，虚表指针 vptr 的初始化时间

首先整理一下虚函数表的特征：

> 虚函数表是全局共享的元素，即全局仅有一个，在编译时就构造完成.
> 虚函数表类似一个数组，类对象中存储 vptr 指针，指向虚函数表，即虚函数表不是函数，不是程序代码，不可能存储在代码段
> 虚函数表存储虚函数的地址,即虚函数表的元素是指向类成员函数的指针,而类中虚函数的个数在编译时期可以确定，即虚函数表的大小可以确定,即大小是在编译时期确定的，不必动态分配内存空间存储虚函数表，所以不在堆中

根据以上特征，虚函数表类似于类中静态成员变量.静态成员变量也是全局共享，大小确定，因此最有可能存在全局数据区，测试结果显示：

虚函数表 vtable 在 Linux/Unix 中存放在可执行文件的只读数据段中(rodata)，这与微软的编译器将虚函数表存放在常量段存在一些差别

由于虚表指针 vptr 跟虚函数密不可分，对于有虚函数或者继承于拥有虚函数的基类，对该类进行实例化时，在构造函数执行时会对虚表指针进行初始化，并且存在对象内存布局的最前面。

一般分为五个区域：栈区、堆区、函数区（存放函数体等二进制代码）、全局静态区、常量区

C++中**虚函数表位于只读数据段（.rodata），也就是 C++内存模型中的常量区；而虚函数则位于代码段（.text），也就是 C++内存模型中的代码区。**

5、模板函数和模板类的特例化

引入原因

编写单一的模板，它能适应多种类型的需求，使每种类型都具有相同的功能，但对于某种特定类型，如果要实现其特有的功能，单一模板就无法做到，这时就需要模板特例化

定义

对单一模板提供的一个特殊实例，它将一个或多个模板参数绑定到特定的类型或值上

（1）模板函数特例化

必须为原函数模板的每个模板参数都提供实参，且使用关键字 template 后跟一个空尖括号对<>，表明将原模板的所有模板参数提供实参，举例如下：

```
template<typename T> //模板函数 int compare(const T &v1,const T &v2){
    if(v1 > v2) return -1;
    if(v2 > v1) return 1;
    return 0;
}//模板特例化,满足针对字符串特定的比较,要提供所有实参,这里只有一个 Ttemplate<>
int compare(const char* const &v1,const char* const &v2){
    return strcmp(p1,p2);
}
```

本质

特例化的本质是实例化一个模板，而非重载它。特例化不影响参数匹配。参数匹配都以最佳匹配为原则。例如，此处如果是 compare(3,5)，则调用普通的模板，若为 compare(“hi”,”haha”)则调用**特例化版本**（因为这个 cosnt char*相对于 T，更匹配实参类型），注意二者函数体的语句不一样了，实现不同功能。

注意

模板及其特例化版本应该声明在同一个头文件中，且所有同名模板的声明应该放在前面，后面放特例化版本。

（2）类模板特例化

原理类似函数模板，**不过在类中，我们可以对模板进行特例化，也可以对类进行部分特例化。**对类进行特例化时，仍然用 template<>表示是一个特例化版本，例如：

```
template<>class hash<sales_data>
{
        size_t operator()(sales_data& s);
        //里面所有T都换成特例化类型版本sales_data
        //按照最佳匹配原则，若T != sales_data，就用普通类模板，否则，就使用含有特定功能的特例化版本。
};
```

类模板的部分特例化

不必为所有模板参数提供实参，可以**指定一部分而非所有模板参数**，一个类模板的部分特例化本身仍是一个模板，使用它时还必须为其特例化版本中未指定的模板参数提供实参(特例化时类名一定要和原来的模板相同，只是参数类型不同，按最佳匹配原则，哪个最匹配，就用相应的模板)

特例化类中的部分成员

可以特例化类中的部分成员函数而不是整个类，举个例子：

```
template<typename T>class Foo
{
    void Bar();
    void Barst(T a)();
};
template<>void Foo<int>::Bar()
{
    //进行int类型的特例化处理
    cout << "我是int型特例化" << endl;
}

Foo<string> fs;
Foo<int> fi;//使用特例化
fs.Bar();//使用的是普通模板，即Foo<string>::Bar()
fi.Bar();//特例化版本，执行Foo<int>::Bar()//Foo<string>::Bar()和Foo<int>::Bar()功能不同
```

6、构造函数、析构函数、虚函数可否声明为内联函数

首先，将这些函数声明为内联函数，在语法上没有错误。因为 inline 同 register 一样，只是个建议，编译器并不一定真正的内联。

> register 关键字：这个关键字请求编译器尽可能的将变量存在 CPU 内部寄存器中，而不是通过内存寻址访问，以提高效率

举个例子：

```
#include <iostream>
using namespace std;
class A
{public:
    inline A() {
                cout << "inline construct()" <<endl;
        }
    inline ~A() {
                cout << "inline destruct()" <<endl;
        }
    inline virtual void  virtualFun() {
                cout << "inline virtual function" <<endl;
        }
};
 int main(){
        A a;
        a.virtualFun();
    return 0;
}
//输出结果
//inline construct()
//inline virtual function
//inline destruct()
```

构造函数和析构函数声明为内联函数是没有意义的

《Effective C++》中所阐述的是：**将构造函数和析构函数声明为 inline 是没有什么意义的，即编译器并不真正对声明为 inline 的构造和析构函数进行内联操作，因为编译器会在构造和析构函数中添加额外的操作（申请/释放内存，构造/析构对象等），致使构造函数/析构函数并不像看上去的那么精简**。

其次，class 中的函数默认是 inline 型的，编译器也只是有选择性的 inline，将构造函数和析构函数声明为内联函数是没有什么意义的。

将虚函数声明为 inline，要分情况讨论

有的人认为虚函数被声明为 inline，但是编译器并没有对其内联，他们给出的理由是 inline 是编译期决定的，而虚函数是运行期决定的，即在不知道将要调用哪个函数的情况下，如何将函数内联呢？

上述观点看似正确，其实不然。

如果虚函数在编译器就能够决定将要调用哪个函数时，就能够内联，那么什么情况下编译器可以确定要调用哪个函数呢，答案是当用对象调用虚函数（此时不具有多态性）时，就内联展开。

综上，当是指向派生类的指针（多态性）调用声明为 inline 的虚函数时，不会内联展开；当是对象本身调用虚函数时，会内联展开，当然前提依然是函数并不复杂的情况下。

7、C++模板是什么，你知道底层怎么实现的？

1. 编译器并不是把函数模板处理成能够处理任意类的函数；编译器从函数模板通过具体类型产生不同的函数；编译器会对函数模板进行两次编译：在声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的代码进行编译。
2. 这是因为函数模板要被实例化后才能成为真正的函数，在使用函数模板的源文件中包含函数模板的头文件，如果该头文件中只有声明，没有定义，那编译器无法实例化该模板，最终导致链接错误。

8、构造函数为什么不能为虚函数？析构函数为什么要虚函数？

1. 从存储空间角度，**虚函数**相应一个指向 vtable 虚函数表的指针，这大家都知道，但是这个指向 vtable 的指针事实上是存储在对象的内存空间的。

问题出来了，假设构造函数是虚的，就须要通过 vtable 来调用，但是对象还没有实例化，也就是内存空间还没有，怎么找 vtable 呢？所以构造函数不能是虚函数。

2. 从使用角度，**虚函数**主要用于在信息不全的情况下，能使重载的函数得到相应的调用。

构造函数本身就是要初始化实例，那使用虚函数也没有实际意义呀。

所以构造函数没有必要是虚函数。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可以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己主动调用的，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。

3. 构造函数不须要是虚函数，也不同意是虚函数，**由于创建一个对象时我们总是要明白指定对象的类型，虽然我们可能通过实验室的基类的指针或引用去訪问它但析构却不一定，我们往往通过基类的指针来销毁对象。

这时候假设析构函数不是虚函数，就不能正确识别对象类型从而不能正确调用析构函数。

4. 从实现上看，**vbtl** 在构造函数调用后才建立，因而构造函数不可能成为虚函数从实际含义上看，在调用构造函数时还不能确定对象的真实类型（由于子类会调父类的构造函数）；并且构造函数的作用是提供初始化，在对象生命期仅仅运行一次，不是对象的动态行为，也没有必要成为虚函数。

5. **当一个构造函数被调用时，它做的首要的事情之中的一个是初始化它的 VPTR。**

因此，它仅仅能知道它是"当前"类的，而全然忽视这个对象后面是否还有继承者。当编译器为这个构造函数产生代码时，它是为这个类的构造函数产生代码——既不是为基类，也不是为它的派生类（由于类不知道谁继承它），所以它使用的 VPTR 必须是对于这个类的 VTABLE。

并且，仅仅要它是最后的构造函数调用，那么在这个对象的生命期内，VPTR 将保持被初始化为指向这个 VTABLE，但假设接着另一个更晚派生的构造函数被调用，这个构造函数又将设置 VPTR 指向它的 VTABLE，等.直到最后的构造函数结束。

VPTR 的状态是由被最后调用的构造函数确定的。这就是为什么构造函数调用是从基类到更加派生类顺序的还有一个理由。

可是，当这一系列构造函数调用正发生时，每一个构造函数都已经设置 VPTR 指向它自己的 VTABLE。

假设函数调用使用虚机制，它将仅仅产生通过它自己的 VTABLE 的调用，而不是最后的 VTABLE（全部构造函数被调用后才会有最后的 VTABLE）。

因为构造函数本来就是为了明确初始化对象成员才产生的，然而 virtual function 主要是为了再不完全了解细节的情况下也能正确处理对象。

另外，virtual 函数是在不同类型的对象产生不同的动作，现在对象还没有产生，如何使用 virtual 函数来完成你想完成的动作。

直接的讲，C++中基类采用 virtual 虚析构函数是**为了防止内存泄漏。**

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。

假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用 virtual 虚析构函数。

9、析构函数的作用，如何起作用？

1、构造函数只是起初始化值的作用，但实例化一个对象的时候，可以通过实例去传递参数，从主函数传递到其他的函数里面，这样就使其他的函数里面有值了。

规则，只要你一实例化对象，系统自动回调用一个构造函数就是你不写，编译器也自动调用一次。

2、析构函数与构造函数的作用相反，用于撤销对象的一些特殊任务处理，可以是释放对象分配的内存空间；特点：析构函数与构造函数同名，但该函数前面加~。

析构函数没有参数，也没有返回值，而且不能重载，在一个类中只能有一个析构函数。 当撤销对象时，编译器也会自动调用析构函数。

每一个类必须有一个析构函数，用户可以自定义析构函数，也可以是编译器自动生成默认的析构函数。一般析构函数定义为类的公有成员。

10、构造函数和析构函数可以调用虚函数吗，为什么

1. 在 C++中，提倡不在构造函数和析构函数中调用虚函数；
2. 构造函数和析构函数调用虚函数时都不使用动态联编，如果在构造函数或析构函数中调用虚函数，则运行的是为构造函数或析构函数自身类型定义的版本；
3. 因为父类对象会在子类之前进行构造，此时子类部分的数据成员还未初始化，因此调用子类的虚函数时不安全的，故而 C++不会进行动态联编；
4. 析构函数是用来销毁一个对象的，在销毁一个对象时，先调用子类的析构函数，然后再调用基类的析构函数。所以在调用基类的析构函数时，派生类对象的数据成员已经销毁，这个时候再调用子类的虚函数没有任何意义。

11、构造函数、析构函数的执行顺序？构造函数和拷贝构造的内部都干了啥？

1) 构造函数顺序

① 基类构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是它们在成员初始化表中的顺序。

② 成员类对象构造函数。如果有多个成员类对象则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们出现在成员初始化表中的顺序。

③ 派生类构造函数。

2) 析构函数顺序

① 调用派生类的析构函数；

② 调用成员类对象的析构函数；

③ 调用基类的析构函数。

12、虚析构函数的作用，父类的析构函数是否要设置为虚函数？

1. C++中基类采用 virtual 虚析构函数是为了防止内存泄漏。

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。

假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。

那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用 virtual 虚析构函数。

1. 纯虚析构函数一定得定义，因为每一个派生类析构函数会被编译器加以扩张，以静态调用的方式调用其每一个虚基类以及上一层基类的析构函数。

因此，缺乏任何一个基类析构函数的定义，就会导致链接失败，最好不要把虚析构函数定义为纯虚析构函数。

13、构造函数析构函数可否抛出异常

1、C++只会析构已经完成的对象，对象只有在其构造函数执行完毕才算是完全构造妥当。在构造函数中发生异常，控制权转出构造函数之外。

因此，在对象 b 的构造函数中发生异常，对象 b 的析构函数不会被调用，因此会造成内存泄漏。

2、用 auto_ptr 对象来取代指针类成员，便对构造函数做了强化，免除了抛出异常时发生资源泄漏的危机，不再需要在析构函数中手动释放资源；

3、如果控制权基于异常的因素离开析构函数，而此时正有另一个异常处于作用状态，C++会调用 terminate 函数让程序结束；

4、如果异常从析构函数抛出，而且没有在当地进行捕捉，那个析构函数便是执行不全的。如果析构函数执行不全，就是没有完成他应该执行的每一件事情。

14、构造函数一般不定义为虚函数的原因

（1）创建一个对象时需要确定对象的类型，而虚函数是在运行时动态确定其类型的。在构造一个对象时，由于对象还未创建成功，编译器无法知道对象的实际类型

（2）虚函数的调用需要虚函数表指针 vptr，而该指针存放在对象的内存空间中，若构造函数声明为虚函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表 vtable 地址用来调用虚构造函数了

（3）虚函数的作用在于通过父类的指针或者引用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，不可能通过父类或者引用去调用，因此就规定构造函数不能是虚函数。

（4）析构函数一般都要声明为虚函数，这个应该是老生常谈了，这里不再赘述。

15、类什么时候会析构？

1. 对象生命周期结束，被销毁时；
2. delete 指向对象的指针时，或 delete 指向对象的基类类型指针，而其基类虚构函数是虚函数时；
3. 对象 i 是对象 o 的成员，o 的析构函数被调用时，对象 i 的析构函数也被调用。

16、构造函数或者析构函数中可以调用虚函数吗

简要结论：

- 从语法上讲，调用完全没有问题。
- 但是从效果上看，往往不能达到需要的目的。

> 《Effective C++》的解释是： 派生类对象构造期间进入基类的构造函数时，对象类型变成了基类类型，而不是派生类类型。 同样，进入基类析构函数时，对象也是基类类型。

举个例子：

语句 1 讲道理应该体现多态性，执行类 A 中的构造和析构函数，从实验结果来看，语句 1 并没有体现，执行流程是先构造基类，所以先调用基类的构造函数，构造完成再执行 A 自己的构造函数，析构时也是调用基类的析构函数，也就是说构造和析构中调用虚函数并不能达到目的，应该避免

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

class Base
{
public:
    Base()
    {
       Function();
    }

    virtual void Function()
    {
        cout << "Base::Fuction" << endl;
    }
    ~Base()
    {
        Function();
    }
};

class A : public Base
{
public:
    A()
    {
      Function();
    }

    virtual void Function()
    {
        cout << "A::Function" << endl;
    }
    ~A()
    {
        Function();
    }
};

int main()
{
    Base* a = new Base;
    delete a;
    cout << "-------------------------" <<endl
    Base* b = new A;//语句1
    delete b;
}
```

```cpp
//输出结果
//Base::Fuction
//Base::Fuction
//-------------------------
//Base::Fuction
//A::Function
//Base::Fuction
```

17、构造函数的几种关键字

default

default 关键字可以显式要求编译器生成合成构造函数，防止在调用时相关构造函数类型没有定义而报错

```
#include <iostream>
using namespace std;
class CString
{
public:
    CString() = default; //语句1
    //构造函数
    CString(const char* pstr) : _str(pstr){}
    void* operator new() = delete;
    //这样不允许使用new 关键字
    //析构函数
    ~CString(){}public:
     string _str;
};

int main(){
   auto a = new CString(); //语句2
   cout << "Hello World" <<endl;
   return 0;
}
//运行结果//Hello World
```

如果没有加语句 1，语句 2 会报错，表示找不到参数为空的构造函数，将其设置为 default 可以解决这个问题

delete

delete 关键字可以删除构造函数、赋值运算符函数等，这样在使用的时候会得到友善的提示

```
#include <iostream>using namespace std;
class CString
{public:
    void* operator new() = delete;//这样不允许使用new关键字
    //析构函数
    ~CString(){}
};

int main(){
   auto a = new CString(); //语句1
   cout << "Hello World" <<endl;
   return 0;
}
```

在执行语句 1 时，会提示 new 方法已经被删除，如果将 new 设置为私有方法，则会报惨不忍睹的错误，因此使用 delete 关键字可以更加人性化的删除一些默认方法

将虚函数定义为纯虚函数（纯虚函数无需定义，= 0 只能出现在类内部虚函数的声明语句处；当然，也可以为纯虚函数提供定义，函数体可以定义在类的外部也可以定义在内部。

18、构造函数、拷贝构造函数和赋值操作符的区别

构造函数

对象不存在，没用别的对象初始化，在创建一个新的对象时调用构造函数

拷贝构造函数

对象不存在，但是使用别的已经存在的对象来进行初始化

赋值运算符

对象存在，用别的对象给它赋值，这属于重载“=”号运算符的范畴，“=”号两侧的对象都是已存在的

举个例子：

```
#include <iostream>
using namespace std;
class A
{
public:
        A()
        {
                cout << "我是构造函数" << endl;
        }
        A(const A& a)
        {
                cout << "我是拷贝构造函数" << endl;
        }
        A& operator = (A& a)
        {
                cout << "我是赋值操作符" << endl;
                return *this;
        }
        ~A() {};
};
int main(){
        A a1; //调用构造函数
        A a2 = a1; //调用拷贝构造函数
        a2 = a1; //调用赋值操作符
        return 0;
}
//输出结果
//我是构造函数
//我是拷贝构造函数
//我是赋值操作符
```

19、拷贝构造函数和赋值运算符重载的区别？

1. 拷贝构造函数是函数，赋值运算符是运算符重载。
2. 拷贝构造函数会生成新的类对象，赋值运算符不能。
3. 拷贝构造函数是直接构造一个新的类对象，所以在初始化对象前不需要检查源对象和新建对象是否相同；赋值运算符需要上述操作并提供两套不同的复制策略，另外赋值运算符中如果原来的对象有内存分配则需要先把内存释放掉。
4. 形参传递是调用拷贝构造函数（调用的被赋值对象的拷贝构造函数），但并不是所有出现"="的地方都是使用赋值运算符，如下：

```
Student s;
Student s1 = s;    // 调用拷贝构造函数
Student s2;
s2 = s;    // 赋值运算符操作
```

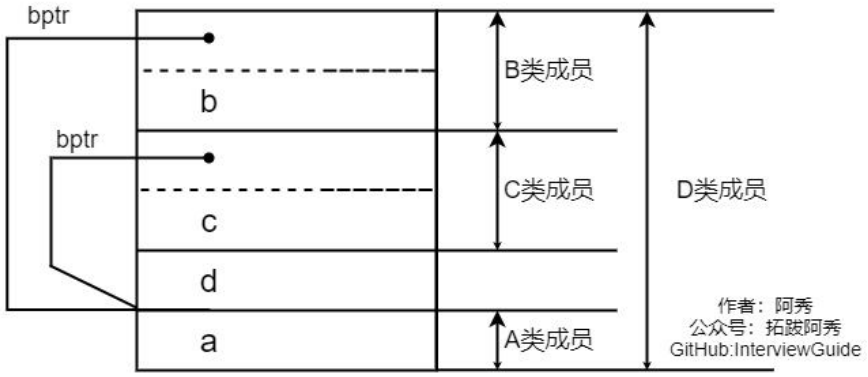
注：类中有指针变量时要重写析构函数、拷贝构造函数和赋值运算符。

20、什么是虚拟继承

由于C++支持多继承，除了public、protected和private三种继承方式外，还支持虚拟（virtual）继承，举个例子：

```
#include <iostream>
using namespace std;
class A{}class B : virtual public A{};
class C : virtual public A{};
class D : public B, public C{};
int main(){
    cout << "sizeof(A): " << sizeof A <<endl;
    // 1，空对象，只有一个占位
    cout << "sizeof(B): " << sizeof B <<endl;
    // 4，一个bptr指针，省去占位,不需要对齐
    cout << "sizeof(C): " << sizeof C <<endl;
    // 4，一个bptr指针，省去占位,不需要对齐
    cout << "sizeof(D): " << sizeof D <<endl;
    // 8，两个bptr，省去占位,不需要对齐
}
```

上述代码所体现的关系是，B 和 C 虚拟继承 A，D 又公有继承 B 和 C，这种方式是一种**菱形继承或者钻石继承**，可以用如下图来表示



虚拟继承的情况下，无论基类被继承多少次，只会存在一个实体。

虚拟继承基类的子类中，子类会增加某种形式的指针，或者指向虚基类子对象，或者指向一个相关的表格；表格中存放的不是虚基类子对象的地址，就是其偏移量，此类指针被称为 bptr，如上图所示。

如果既存在 vptr 又存在 bptr，某些编译器会将其优化，合并为一个指针。

21、什么情况会自动生成默认构造函数？

1、带有默认构造函数的类成员对象，如果一个类没有任何构造函数，但它含有一个成员对象，而后者有默认构造函数，那么编译器就为该类合成出一个默认构造函数。

不过这个合成操作只有在构造函数真正被需要的时候才会发生；

如果一个类 A 含有多个成员类对象的话，那么类 A 的每一个构造函数必须调用每一个成员对象的默认构造函数而且必须按照类对象在类 A 中的声明顺序进行；

2、带有默认构造函数的基类，如果一个没有任务构造函数的派生类派生自一个带有默认构造函数基类，那么该派生类会合成一个构造函数调用上一层基类的默认构造函数；

3、带有一个虚函数的类

4、带有一个虚基类的类

5、合成的默认构造函数中，只有基类子对象和成员类对象会被初始化。所有其他的非静态数据成员都不会被初始化。

22、抽象基类为什么不能创建对象？

抽象类是一种特殊的类，它是为了抽象和设计的目的为建立的，它处于继承层次结构的较上层。

1、抽象类的定义：

称带有纯虚函数的类为抽象类。

2、抽象类的作用：

抽象类的主要作用是将有关的操作作为结果接口组织在一个继承层次结构中，由它来为派生类提供一个公共的根，派生类将具体实现在其基类中作为接口的操作。

所以派生类实际上刻画了一组子类的操作接口的通用语义，这些语义也传给子类，子类可以具体实现这些语义，也可以再将这些语义传给自己的子类。

3、 抽象类只能作为基类来使用

其纯虚函数的实现由派生类给出。如果派生类中没有重新定义纯虚函数，而只是继承基类的纯虚函数，则这个派生类仍然还是一个抽象类。

如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体的类。

抽象类是不能定义对象的。一个纯虚函数不需要（但是可以）被定义。

4、纯虚函数定义

纯虚函数是一种特殊的虚函数，它的一般格式如下：

```
class <类名>   {   virtual <类型><函数名>(<参数表>)=0;   …   };
```

在许多情况下，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，它的实现留给该基类的派生类去做。这就是纯虚函数的作用。

纯虚函数可以让类先具有一个操作名称，而没有操作内容，让派生类在继承时再去具体地给出定义。凡是含有纯虚函数的类叫做抽象类。

这种类不能声明对象，只是作为基类为派生类服务。除非在派生类中完全实现基类中所有的的纯虚函数，否则，派生类也变成了抽象类，不能实例化对象。

5、纯虚函数引入原因

1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。 2、在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔 雀等子类，但动物本身生成对象明显不合常理。

为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;）。若要使派生类为非抽象类，则编译器要求在派生类中，必须对纯虚函数予以重载以实现多态性。同时含有纯虚函数的类称为抽象类，它不能生成对象。这样就很好地解决了上述两个问题。

例如，绘画程序中，shape 作为一个基类可以派生出圆形、矩形、正方形、梯形等， 如果我要求面积总和的话，那么会可以使用一个 shape * 的数组，只要依次调用派生类的 area() 函数了。如果不用接口就没法定义成数组，因为既可以是 circle ,也可以是 square ,而且以后还可能加上 rectangle，等等.

6、相似概念

1、多态性

指相同对象收到不同消息或不同对象收到相同消息时产生不同的实现动作。C++支持两种多态性：编译时多态性，运行时多态性。

- a.编译时多态性：通过重载函数实现
- b.运行时多态性：通过虚函数实现。

2、虚函数

虚函数是在基类中被声明为 virtual，并在派生类中重新定义的成员函数，可实现成员函数的动态重载。

3、抽象类

包含纯虚函数的类称为抽象类。由于抽象类包含了没有定义的纯虚函数，所以不能定义抽象类的对象。

23、模板类和模板函数的区别是什么？

函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。即函数模板允许隐式调用和显式调用而类模板只能显示调用。在使用时类模板必须加<T>，而函数模板不必

24、多继承的优缺点，作为一个开发者怎么看待多继承

1. C++允许为一个派生类指定多个基类，这样的继承结构被称做多重继承。
2. 多重继承的优点很明显，就是对象可以调用多个基类中的接口；
3. 如果派生类所继承的多个基类有相同的基类，而派生类对象需要调用这个祖先类的接口方法，就会容易出现二义性
4. 加上全局符确定调用哪一份拷贝。比如 pa.Author::eat()调用属于 Author 的拷贝。
5. 使用虚拟继承，使得多重继承类 Programmer_Author 只拥有 Person 类的一份拷贝。

25、模板和实现可不可以不写在一个文件里面？为什么？

因为在编译时模板并不能生成真正的二进制代码，而是在编译调用模板类或函数的 CPP 文件时才会去找对应的模板声明和实现。在这种情况下编译器是不知道实现模板类或函数的 CPP 文件的存在，所以它只能找到模板类或函数的声明而找不到实现，而只好创建一个符号寄希望于链接程序找地址。

但模板类或函数的实现并不能被编译成二进制代码，结果链接程序找不到地址只好报错了。

《C++编程思想》第 15 章(第 300 页)说明了原因：模板定义很特殊。由 template<...>处理的任何东西都意味着编译器在当时不为它分配存储空间，

它一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板的多重定义。所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

26、将字符串“hello world”从开始到打印到屏幕上的全过程?

1.用户告诉操作系统执行 HelloWorld 程序（通过键盘输入等）

2．操作系统：找到 helloworld 程序的相关信息，检查其类型是否是可执行文件；并通过程序首部信息，确定代码和数据在可执行文件中的位置并计算出对应的磁盘块地址。

3．操作系统：创建一个新进程，将 HelloWorld 可执行文件映射到该进程结构，表示由该进程执行 helloworld 程序。

4．操作系统：为 helloworld 程序设置 cpu 上下文环境，并跳到程序开始处。

5．执行 helloworld 程序的第一条指令，发生缺页异常

6．操作系统：分配一页物理内存，并将代码从磁盘读入内存，然后继续执行 helloworld 程序

7．helloword 程序执行 puts 函数（系统调用），在显示器上写一字符串

8．操作系统：找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以，操作系统将要写的字符串送给该进程

9．操作系统：控制设备的进程告诉设备的窗口系统，它要显示该字符串，窗口系统确定这是一个合法的操作，然后将字符串转换成像素，将像素写入设备的存储映像区

10．视频硬件将像素转换成显示器可接收和一组控制数据信号

11．显示器解释信号，激发液晶屏

12．OK，我们在屏幕上看到了 HelloWorld

27、为什么拷贝构造函数必须传引用不能传值？

拷贝构造函数的作用就是用来复制对象的，在使用这个对象的实例来初始化这个对象的一个新的实例。

参数传递过程到底发生了什么？

将地址传递和值传递统一起来，归根结底还是传递的是"值"(地址也是值，只不过通过它可以找到另一个值)！

a 值传递:

对于内置数据类型的传递时，直接赋值拷贝给形参(注意形参是函数内局部变量)； 对于类类型的传递时，需要首先调用该类的拷贝构造函数来初始化形参(局部对象)；

如 void foo(class_type obj_local){}, 如果调用 foo(obj); 首先 class_type obj_local(obj) ,这样就定义了局部变量 obj_local 供函数内部使用

b 引用传递:

无论对内置类型还是类类型，传递引用或指针最终都是传递的地址值！而地址总是指针类型(属于简单类型), 显然参数传递时，按简单类型的赋值拷贝，而不会有拷贝构造函数的调用(对于类类型). 上述 1) 2)回答了为什么拷贝构造函数使用值传递会产生无限递归调用，内存溢出。

拷贝构造函数用来初始化一个非引用类类型对象，如果用传值的方式进行传参数，那么构造实参需要调用拷贝构造函数，而拷贝构造函数需要传递实参，所以会一直递归。

28、静态函数能定义为虚函数吗？常函数呢？说说你的理解

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。

29、虚函数的代价是什么？

1. 带有虚函数的类，每一个类会产生一个虚函数表，用来存储指向虚成员函数的指针，增大类；
2. 带有虚函数的类的每一个对象，都会有有一个指向虚表的指针，会增加对象的空间大小；
3. 不能再是内联的函数，因为内联函数在编译阶段进行替代，而虚函数表示等待，在运行阶段才能确定到低是采用哪种函数，虚函数不能是内联函数。

30、说一说你了解到的移动构造函数？

有时候我们会遇到这样一种情况，我们用对象 a 初始化对象 b 后对象 a 我们就不在使用了，但是对象 a 的空间还在呀（在析构之前），既然拷贝构造函数，实际上就是把 a 对象的内容复制一份到 b 中，那么为什么我们不能直接使用 a 的空间呢？这样就避免了新的空间的分配，大大降低了构造的成本。这就是移动构造函数设计的初衷；

拷贝构造函数中，对于指针，我们一定要采用深层复制，而移动构造函数中，对于指针，我们采用浅层复制；

C++引入了移动构造函数，专门处理这种，用 a 初始化 b 后，就将 a 析构的情况；

与拷贝类似，移动也使用一个对象的值设置另一个对象的值。但是，又与拷贝不同的是，移动实现的是对象值真实的转移（源对象到目的对象）：源对象将丢失其内容，其内容将被目的对象占有。移动操作的发生的时候，是当移动值的对象是未命名的对象的时候。这里未命名的对象就是那些临时变量，甚至都不会有名称。典型的未命名对象就是函数的返回值或者类型转换的对象。使用临时对象的值初始化另一个对象值，不会要求对对象的复制：因为临时对象不会有其它使用，因而，它的值可以被移动到目的对象。做到这些，就要使用移动构造函数和移动赋值：当使用

一个临时变量对象进行构造初始化的时候，调用移动构造函数。类似的，使用未命名的变量的值赋给一个对象时，调用移动赋值操作；

```
Example6 (Example6&& x) : ptr(x.ptr)
  {
    x.ptr = nullptr;
  }

  // move assignment
  Example6& operator= (Example6&& x)
  {
   delete ptr;
   ptr = x.ptr;
   x.ptr=nullptr;
    return *this;
}
```

31、 什么情况下会合成构造函数？都说一说，你知道的都说一下

1. 如果一个类没有任何构造函数，但他含有一个成员对象，该成员对象含有默认构造函数，那么编译器就为该类合成一个默认构造函数，因为不合成一个默认构造函数那么该成员对象的构造函数不能调用；
2. 没有任何构造函数的类派生自一个带有默认构造函数的基类，那么需要为该派生类合成一个构造函数，只有这样基类的构造函数才能被调用；
3. 带有虚函数的类，虚函数的引入需要进入虚表，指向虚表的指针，该指针是在构造函数中初始化的，所以没有构造函数的话该指针无法被初始化；
4. 带有一个虚基类的类

还有一点需要注意的是：

1. 并不是任何没有构造函数的类都会合成一个构造函数
2. 编译器合成出来的构造函数并不会显示设定类内的每一个成员变量

32、那什么时候需要合成拷贝构造函数呢？

有三种情况会以一个对象的内容作为另一个对象的初值：

1、对一个对象做显示的初始化操作，X xx = x;

2、当对象被当做参数交给某个函数时；

3、当函数传回一个类对象时；

- 如果一个类没有拷贝构造函数，但是含有一个类类型的成员变量，该类型含有拷贝构造函数，此时编译器会为该类合成一个拷贝构造函数；
- 如果一个类没有拷贝构造函数，但是该类继承自含有拷贝构造函数的基类，此时编译器会为该类合成一个拷贝构造函数；
- 如果一个类没有拷贝构造函数，但是该类声明或继承了虚函数，此时编译器会为该类合成一个拷贝构造函数；
- 如果一个类没有拷贝构造函数，但是该类含有虚基类，此时编译器会为该类合成一个拷贝构造函数；

33、构造函数的执行顺序是什么？

1. 在派生类构造函数中，所有的虚基类及上一层基类的构造函数调用；
2. 对象的 vptr 被初始化；
3. 如果有成员初始化列表，将在构造函数体内扩展开来，这必须在 vptr 被设定之后才做；
4. 执行程序员所提供的代码；

34、一个类中的全部构造函数的扩展过程是什么？

1. 记录在成员初始化列表中的数据成员初始化操作会被放在构造函数的函数体内，并与成员的声明顺序为顺序；
2. 如果一个成员并没有出现在成员初始化列表中，但它有一个默认构造函数，那么默认构造函数必须被调用；
3. 如果 class 有虚表，那么它必须被设定初值；
4. 所有上一层的基类构造函数必须被调用；
5. 所有虚基类的构造函数必须被调用。

35、哪些函数不能是虚函数？把你知道的都说一说

1. 构造函数，构造函数初始化对象，派生类必须知道基类函数干了什么，才能进行构造；当有虚函数时，每一个类有一个虚表，每一个对象有一个虚表指针，虚表指针在构造函数中初始化；
2. 内联函数，内联函数表示在编译阶段进行函数体的替换操作，而虚函数意味着在运行期间进行类型确定，所以内联函数不能是虚函数；
3. 静态函数，静态函数不属于对象属于类，静态成员函数没有 this 指针，因此静态函数设置为虚函数没有任何意义。
4. 友元函数，友元函数不属于类的成员函数，不能被继承。对于没有继承特性的函数没有虚函数的说法。
5. 普通函数，普通函数不属于类的成员函数，不具有继承特性，因此普通函数没有虚函数

36、什么是纯虚函数，与虚函数的区别

虚函数和纯虚函数区别？

- 虚函数是为了实现动态编联产生的，目的是通过基类类型的指针指向不同对象时，自动调用相应的、和基类同名的函数（使用同一种调用形式，既能调用派生类又能调用基类的同名函数）。虚函数需要在基类中加上 virtual 修饰符修饰，因为 virtual 会被隐式继承，所以子类中相同函数都是虚函数。当一个成员函数被声明为虚函数之后，其派生类中同名函数自动成为虚函数，在派生类中重新定义此函数时要求函数名、返回值类型、参数个数和类型全部与基类函数相同。
- 纯虚函数只是相当于一个接口名，但含有纯虚函数的类不能够实例化。

纯虚函数首先是虚函数，其次它没有函数体，取而代之的是用“=0”。

既然是虚函数，它的函数指针会被存在虚函数表中，由于纯虚函数并没有具体的函数体，因此它在虚函数表中的值就为 0，而具有函数体的虚函数则是函数的具体地址。

一个类中如果有纯虚函数的话，称其为抽象类。抽象类不能用于实例化对象，否则会报错。抽象类一般用于定义一些公有的方法。子类继承抽象类也必须实现其中的纯虚函数才能实例化对象。

举个例子：

```
#include <iostream>using namespace std;
class Base
{public:
        virtual void fun1()
        {
                cout << "普通虚函数" << endl;
        }
        virtual void fun2() = 0;
        virtual ~Base() {}
};
class Son : public Base
{public:
        virtual void fun2()
        {
                cout << "子类实现的纯虚函数" << endl;
        }
};
int main(){
        Base* b = new Son;
        b->fun1(); //普通虚函数
        b->fun2(); //子类实现的纯虚函数
        return 0;
}
```