**C++中为什么存在this指针？**

成员函数存储在公共的代码段，不同的对象调用成员函数时候，那么编译器如何知道具体操作的是哪一个对象呢？在这里就引出了this指针。

C++编译器给每个 “非静态的成员函数” 增加了一个隐藏的指针参数，让该指针指向当前对象(函数运行时调用该函数的对象)，在函数体中所有成员变量的操作，都是通过这个指针来完成的。只不过所有的操作对用户来说都是透明的，不需要用户来传递，编译器自动完成。

this指针特性：

1、只能在“成员函数”中使用

2、this指针类型： 类类型 \*const （加const是为了保证，指针的指向不被更改）

3、this指针不存储在对象中，不影响对象大小，且始终指向当前对象

4、this指针是“成员函数”的第一个隐藏参数，由编译器自动给出

5、主要是通过ecx寄存器来传递（并不是所有的）

## 构造函数可以成为虚函数吗？

不可以。因为在构造函数执行结束后，虚函数表指针才会被正确的初始化。

在c++ 的多态中，虚函数表是由编译器自动生成与维护的，虚函数表指针是由构造函数初始化完成的，即构造函数相当于是虚函数的入口点，负责调用虚函数的前期工作；在构造函数执行的过程中，虚函数表指针有可能未被正确的初始化；由于在不同的c++编译器中，虚函数表 与 虚函数表指针的实现有所不同，所以禁止将构造函数声明为虚函数。

### 虚表指针初始化时机

由于每个对象调用的虚函数都是通过虚表指针来索引的，也就决定了虚表指针的正确初始化是非常重要的。换句话说，在虚表指针没有正确初始化之前，我们不能够去调用虚函数。那么虚表指针在什么时候，或者说在什么地方初始化呢？

答案是在构造函数中进行虚表的创建和虚表指针的初始化。还记得构造函数的调用顺序吗，在构造子类对象时，要先调用父类的构造函数，此时编译器只“看到了”父类，并不知道后面是否后还有继承者，它初始化父类对象的虚表指针，该虚表指针指向父类的虚表。当执行子类的构造函数时，子类对象的虚表指针被初始化，指向自身的虚表。

### 虚函数表存放位置

1、虚函数表是class specific的，也就是针对一个类来说的，这里有点像一个类里面的staic成员变量，即它是属于一个类所有对象的，不是属于某一个对象特有的，是一个类所有对象共有的。

2、虚函数表是编译器来选择实现的，编译器的种类不同，可能实现方式不一样，就像前面我们说的vptr在一个对象的最前面，但是也有其他实现方式，不过目前gcc 和微软的编译器都是将vptr放在对象内存布局的最前面。

3、虽然我们知道vptr指向虚函数表，那么虚函数表具体存放在内存哪个位置呢，虽然这里我们已经可以得到虚函数表的地址。实际上虚函数指针是在构造函数执行时初始化的，而虚函数表是存放在可执行文件中的。下面的一篇博客测试了微软的编译器将虚函数表存放在了目标文件或者可执行文件的常量段中，<http://blog.csdn.net/vicness/article/details/3962767>，不过我在gcc下的汇编文件中没有找到vtbl的具体存放位置，主要是对可执行文件的装载和运行原理还没有深刻的理解，相信不久有了这些知识之后会很轻松的找到虚函数表到底存放在目标文件的哪一个段中。

4、经过测试，在gcc编译器的实现中虚函数表vtable存放在可执行文件的只读数据段.rodata中。

**C++ 多态**

子类跟父类一样有一个虚表指针。

子类的虚函数表一部分继承自父类。如果重写了虚函数，那么子类的虚函数会在虚表上覆盖父类的虚函数。

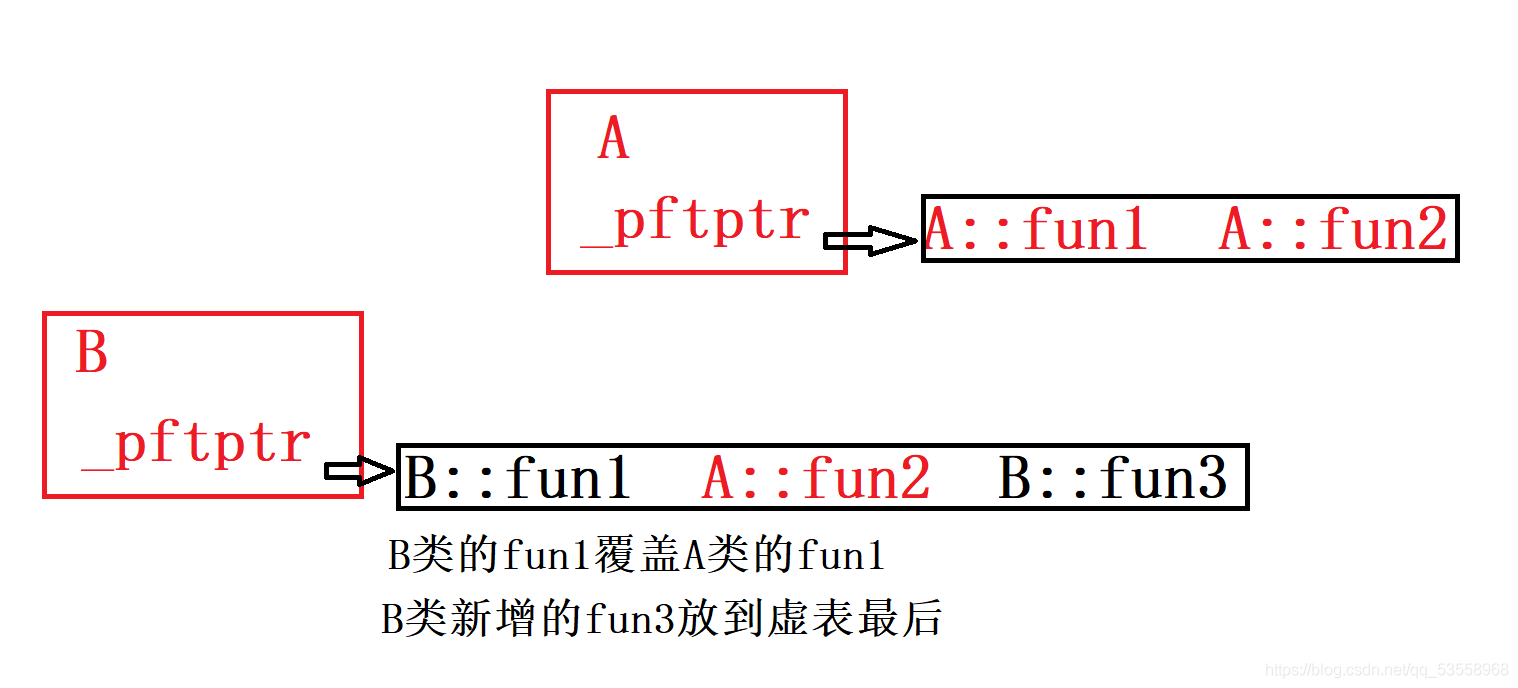
本质上虚函数表是一个虚函数指针数组，最后一个元素是nullptr，代表虚表的结束。

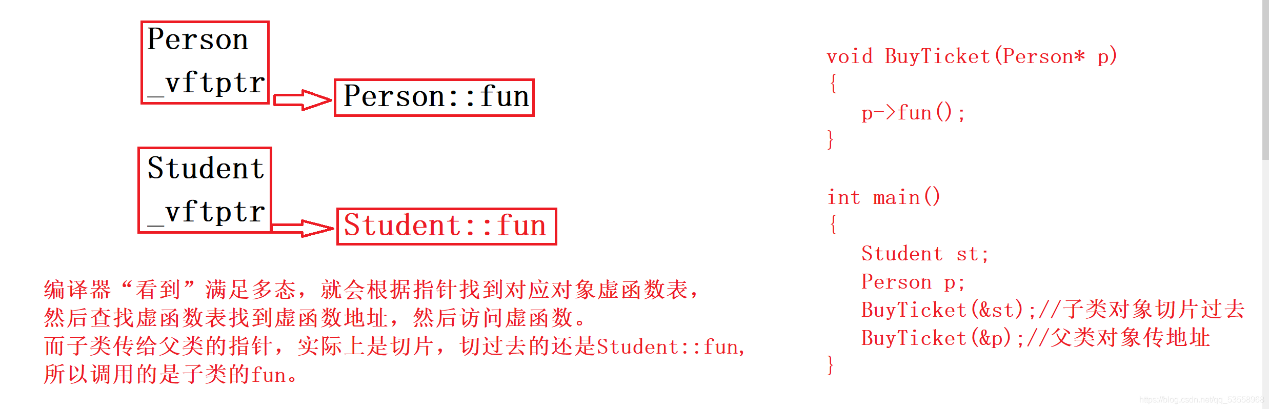
所以，如果继承了虚函数，那么

1 子类先拷贝一份父类虚表，然后用一个虚表指针指向这个虚表。

2 如果有虚函数重写，那么在子类的虚表上用子类的虚函数覆盖。

3 子类新增的虚函数按其在子类中的声明次序增加到子类虚表的最后。





**面向对象的基本特征**

面向对象编程(OOP)的3个基本特征是：封装、继承、多态

        封装：将属性和方法(数据和功能)封装在一起;

     继承：继承是指这样一种能力：它可以使用现有类的功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

        多态：允许让父类的指针分别指向不同的子类, 调用不同子类的同一个方法,  会有不同的执行效果

**C++ 11 新特性**

https://cloud.tencent.com/developer/article/1745592

#### ****auto & decltype****

关于C++11新特性，最先提到的肯定是类型推导，C++11引入了auto和decltype关键字，使用他们可以在编译期就推导出变量或者表达式的类型，方便开发者编码也简化了代码。

* auto：让编译器在编译器就推导出变量的类型，可以通过=右边的类型推导出变量的类型。

auto a = 10; // 10是int型，可以自动推导出a是int

复制

* decltype：相对于auto用于推导变量类型，而decltype则用于推导表达式类型，这里只用于编译器分析表达式的类型，表达式实际不会进行运算。

cont int &i = 1;

int a = 2;

decltype(i) b = 2; // b是const int&

#### ****左值右值****

众所周知C++11新增了右值引用，这里涉及到很多概念：

* 左值：可以取地址并且有名字的东西就是左值。
* 右值：不能取地址的没有名字的东西就是右值。
* 纯右值：运算表达式产生的临时变量、不和对象关联的原始字面量、非引用返回的临时变量、lambda表达式等都是纯右值。
* 将亡值：可以理解为即将要销毁的值。
* 左值引用：对左值进行引用的类型。
* 右值引用：对右值进行引用的类型。
* 移动语义：转移资源所有权，类似于转让或者资源窃取的意思，对于那块资源，转为自己所拥有，别人不再拥有也不会再使用。
* 完美转发：可以写一个接受任意实参的函数模板，并转发到其它函数，目标函数会收到与转发函数完全相同的实参。
* 返回值优化：当函数需要返回一个对象实例时候，就会创建一个临时对象并通过复制构造函数将目标对象复制到临时对象，这里有复制构造函数和析构函数会被多余的调用到，有代价，而通过返回值优化，C++标准允许省略调用这些复制构造函数。

#### ****列表初始化****

在C++11中可以直接在变量名后面加上初始化列表来进行对象的初始化

#### ****std::function & std::bind & lambda表达式****

c++11新增了std::function、std::bind、lambda表达式等封装使函数调用更加方便

#### ****模板的改进****

C++11关于模板有一些细节的改进：

* 模板的右尖括号
* 模板的别名
* 函数模板的默认模板参数

#### ****并发****

c++11关于并发引入了好多好东西，有：

* std::thread相关
* std::mutex相关
* std::lock相关
* std::atomic相关
* std::call\_once相关
* volatile相关
* std::condition\_variable相关
* std::future相关
* async相关

#### ****智能指针****

很多人谈到c++，说它特别难，可能有一部分就是因为c++的内存管理吧，不像java那样有虚拟机动态的管理内存，在程序运行过程中可能就会出现内存泄漏，然而这种问题其实都可以通过c++11引入的智能指针来解决，相反我还认为这种内存管理还是c++语言的优势，因为尽在掌握。

c++11引入了三种智能指针：

* std::shared\_ptr
* std::weak\_ptr
* std::unique\_ptr

#### ****基于范围的for循环****

直接看代码

vector<int> vec;

for (auto iter = vec.begin(); iter != vec.end(); iter++) { // before c++11

cout << \*iter << endl;

}

for (int i : vec) { // c++11基于范围的for循环

cout << "i" << endl;

}

#### ****委托构造函数****

委托构造函数允许在同一个类中一个构造函数调用另外一个构造函数，可以在变量初始化时简化操作，通过代码来感受下委托构造函数的妙处吧：

不使用委托构造函数：

struct A {

A(){}

A(int a) { a\_ = a; }

A(int a, int b) { // 好麻烦

a\_ = a;

b\_ = b;

}

A(int a, int b, int c) { // 好麻烦

a\_ = a;

b\_ = b;

c\_ = c;

}

int a\_;

int b\_;

int c\_;

};

复制

使用委托构造函数：

struct A {

A(){}

A(int a) { a\_ = a; }

A(int a, int b) : A(a) { b\_ = b; }

A(int a, int b, int c) : A(a, b) { c\_ = c; }

int a\_;

int b\_;

int c\_;

};

复制

初始化变量是不是方便了许多。

#### ****继承构造函数****

继承构造函数可以让派生类直接使用基类的构造函数，如果有一个派生类，我们希望派生类采用和基类一样的构造方式，可以直接使用基类的构造函数，而不是再重新写一遍构造函数，老规矩，看代码：

不使用继承构造函数：

struct Base {

Base() {}

Base(int a) { a\_ = a; }

Base(int a, int b) : Base(a) { b\_ = b; }

Base(int a, int b, int c) : Base(a, b) { c\_ = c; }

int a\_;

int b\_;

int c\_;

};

struct Derived : Base {

Derived() {}

Derived(int a) : Base(a) {} // 好麻烦

Derived(int a, int b) : Base(a, b) {} // 好麻烦

Derived(int a, int b, int c) : Base(a, b, c) {} // 好麻烦

};

int main() {

Derived a(1, 2, 3);

return 0;

}

复制

使用继承构造函数：

struct Base {

Base() {}

Base(int a) { a\_ = a; }

Base(int a, int b) : Base(a) { b\_ = b; }

Base(int a, int b, int c) : Base(a, b) { c\_ = c; }

int a\_;

int b\_;

int c\_;

};

struct Derived : Base {

using Base::Base;

};

int main() {

Derived a(1, 2, 3);

return 0;

}

复制

只需要使用using Base::Base继承构造函数，就免去了很多重写代码的麻烦。

#### ****nullptr****

nullptr是c++11用来表示空指针新引入的常量值，在c++中如果表示空指针语义时建议使用nullptr而不要使用NULL，因为NULL本质上是个int型的0，其实不是个指针。举例：

void func(void \*ptr) {

cout << "func ptr" << endl;

}

void func(int i) {

cout << "func i" << endl;

}

int main() {

func(NULL); // 编译失败，会产生二义性

func(nullptr); // 输出func ptr

return 0;

}

**左值、右值、移动语义**

左值：等号左边，可以取地址，有名字的值（函数名、变量名、前置自增表达式）

右值：等号右边，不可取地址，无名字的值（返回非引用类型的函数调用、后置自增表达式）

左值引用：等号右边的值必须可以取地址或者是常量引用

右值引用：等号右边必须是右值

移动语义：转移所有权，对于要操作的资源，转为自己所有，别人不再拥有也不再使用

class A {

public:

A(int size) : size\_(size) {

data\_ = new int[size];

}

A() {}

A(const A& a) {

size\_ = a.size\_;

data\_ = new int[size\_];

cout << "copy " << endl;

}

A(A&& a) {

this->data\_ = a.data\_;

a.data\_ = nullptr;

cout << "move " << endl;

}

~A() {

if (data\_ != nullptr) {

delete[] data\_;

}

}

int \*data\_;

int size\_;

};

int main() {

A a(10);

A b = a;

A c = std::move(a); // 调用移动构造函数

return 0;

}

**Lambda 表达式**

[ ] : 不捕获变量

[ & ] : 捕获作用域外所有变量，按引用使用

[ = ] : 捕获作用域外所有变量，在作用域内使用其副本，不可改变值

[ =, &a ] ：捕获作用域外所有变量，在作用域内使用其副本，不可改变值，变量 a 按引用捕获

[ a ] : 只捕获变量 a

[ this ] : 捕获 this 指针