

C++虚函数表深入探索(详细全面)

发布于2020-03-16 23:43:31

阅读 8K

这篇博客可能有一点点长，代码也有一点点多，但是仔细阅读分析完，会对虚函数表有一个深刻的认识。

什么是虚函数表？

对于一个类来说，如果类中存在虚函数，那么该类的大小就会多4个字节，而这4个字节就是一个指针的大小，这个指针指向虚函数表。所以，如果对象存在虚函数，那么编译器就会生成一个指向虚函数表的指针，所有的虚函数都存在于这个表中，虚函数表就可以理解为一个数组，每个单元用来存放虚函数的地址。

虚函数（Virtual Function）是通过一张虚函数表来实现的。简称为V-Table。在这个表中，主要是一个类的虚函数的地址表，这张表解决了继承、覆盖的问题，保证其真实反应实际的函数。这样，在有虚函数的类的实例中分配了指向这个表的指针的内存，所以，当用父类的指针来操作一个子类的时候，这张虚函数表就显得尤为重要了，它就像一个地图一样，指明了实际所应该调用的函数。

-----百度百

科

虚函数表存在的位置

由于虚函数表是由编译器给我们生成的，那么编译器会把虚函数表安插在哪个位置呢？下面可以简单的写一个示例来证明一下虚函数表的存在，以及观察它所存在的位置，先来看一下代码：

```
1 | #include <iostream>
2 | #include <stdio.h>
3 | using namespace std;
4 |
5 | class A{
6 | public:
7 |     int x;
8 |     virtual void b() {}
9 | };
10 |
11 | int main()
12 | {
13 |     A* p = new A;
14 |     cout << p << endl;
15 |     cout << &p->x << endl;
16 |     return 0;
17 | }
```

定义了一个类A，含有一个x和一个虚函数，实例化一个对象，然后输出对象的地址和对象成员x的地址，我们想一下，如果对象的地址和x的地址相同，那么就意味着编译器把虚函数表放在了末尾，如果两个地址不同，那么就意味着虚函数表是放在最前面的。运行结果为16进制，然后我们把它转为10进制观察一下：

作者介绍



Ch_Zaqdt

关注

专栏

文章	阅读量	获赞	作者排名
363	178.3K	1K	840

精选专题

腾讯云原生专题
云原生技术干货，业务实践落地。

活动推荐

视频公开课上线啦
Vite学习指南，基于腾讯云Webify部署项目

立即查看

腾讯云自媒体分享计划
入驻云加社区，共享百万资源包。

立即入驻

运营活动

腾讯云

广告

OCR文字识别

新用户9.9元专享价

立即抢购

目录

- 什么是虚函数表？
- 虚函数表存在的位置
- 获取虚函数表
- 多重继承的虚函数表：
- 虚函数指针和虚函数表的创建时机：
- 虚函数表的深入探索：

专栏

问答

沙龙

更多



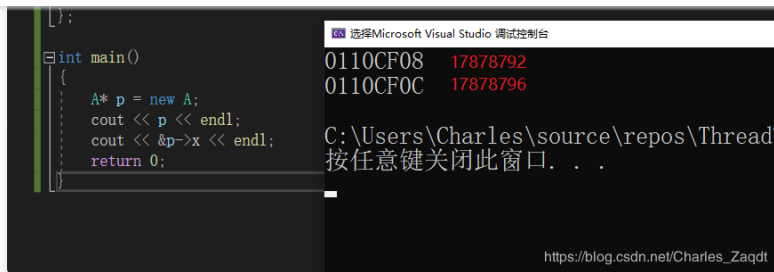
找文章 / 找答案 / 找技术大牛

创作

提问

登录

注册



可以观察到结果是不同的，而且正好相差了4个字节，由此可见，编译器把生成的虚函数表放在了最前面。

获取虚函数表

既然虚函数表是真实存在的，那么我们能不能想办法获取到虚函数表呢？其实我们可以通过指针的形式去获得，因为前面也提到了，我们可以把虚函数表看作是一个数组，每一个单元用来存放虚函数的地址，那么当调用的时候可以直接通过指针去调用所需要的函数就行了。我们就类比这个思路，去获取一下虚函数表。首先定义两个类，一个是基类一个是派生类，代码如下：

```
1 class Base {
2 public:
3     virtual void a() { cout << "Base a()" << endl; }
4     virtual void b() { cout << "Base b()" << endl; }
5     virtual void c() { cout << "Base c()" << endl; }
6 };
7
8 class Derive : public Base {
9 public:
10     virtual void b() { cout << "Derive b()" << endl; }
11 };
```

现在我们设想一下Derive类中的虚函数表是什么样的，它应该是含有三个指针，分别指向基类的虚函数a和基类的虚函数c和自己的虚函数b(因为基类和派生类中含有同名函数，被覆盖)，那么我们就用下面的方式来验证一下：

```
1 Derive* p = new Derive;
2 long* tmp = (long*)p; // 先将p强制转换为long:
3
4 // 由于tmp是虚函数表指针，那么*tmp就是虚函数表
5 long* vptr = (long*)(*tmp);
6 for (int i = 0; i < 3; i++) {
7     printf("vptr[%d] : %p\n", i, vptr[i]);
8 }
```

同理，我们把基类的虚函数表的内容也用这种方法获取出来，然后二者进行比较一下，看看是否符合我们上面所说的那个情况。先看一下完整的代码：

```
1 #include <iostream>
2 #include <stdio.h>
3 using namespace std;
4
5 class Base {
6 public:
7     virtual void a() { cout << "Base a()" << endl; }
8     virtual void b() { cout << "Base b()" << endl; }
9     virtual void c() { cout << "Base c()" << endl; }
10 };
11
12 class Derive : public Base {
13 public:
14     virtual void b() { cout << "Derive b()" << endl; }
15 };
16
```

专栏

问答

沙龙

更多

玩转C++
有奖调研

找文章 / 找答案 / 找技术大牛

创作

提问

登录

注册

```

20 Base* q = new Base;
21 long* tmp1 = (long*)q;
22 long* vptr1 = (long*)(*tmp1);
23 for (int i = 0; i < 3; i++) {
24     printf("vptr[%d] : %p\n", i, vptr1[i]);
25 }
26
27 Derive* p = new Derive;
28 long* tmp = (long*)p;
29 long* vptr = (long*)(*tmp);
30 cout << "-----Derive-----" << endl;
31 for (int i = 0; i < 3; i++) {
32     printf("vptr[%d] : %p\n", i, vptr[i]);
33 }
34 return 0;
35 }

```

运行结果如下图所示:

```

t main()
cout << "-----Base-----" << endl;
Base* q = new Base;
long* tmp1 = (long*)q;
long* vptr1 = (long*)(*tmp1);
for (int i = 0; i < 3; i++) {
    printf("vptr[%d] : %p\n", i, vptr1[i]);
}

Derive* p = new Derive;
long* tmp = (long*)p;
long* vptr = (long*)(*tmp);
cout << "-----Derive-----" << endl;
for (int i = 0; i < 3; i++) {
    printf("vptr[%d] : %p\n", i, vptr[i]);
}

```

Microsoft Visual Studio 调试控制台

```

-----Base-----
vptr[0] : 00D61032
vptr[1] : 00D611D6
vptr[2] : 00D612A8
-----Derive-----
vptr[0] : 00D61032
vptr[1] : 00D610DC
vptr[2] : 00D612A8
C:\Users\Charles\source\repos\ThreadTest
按任意键关闭此窗口...
https://blog.csdn.net/Charles_Zaadt

```

可见基类中的三个指针分别指向a,b,c虚函数, 而派生类中的三个指针中第一个和第三个和基类中的相同, 那么这就印证了上述我们所假设的情况, 那么这也就是虚函数表。但是仅仅只是观察指向的地址, 还不是让我们观察的特别清楚, 那么我们就通过定义函数指针, 来调用一下这几个地址, 看看结果是什么样的, 下面直接上代码:

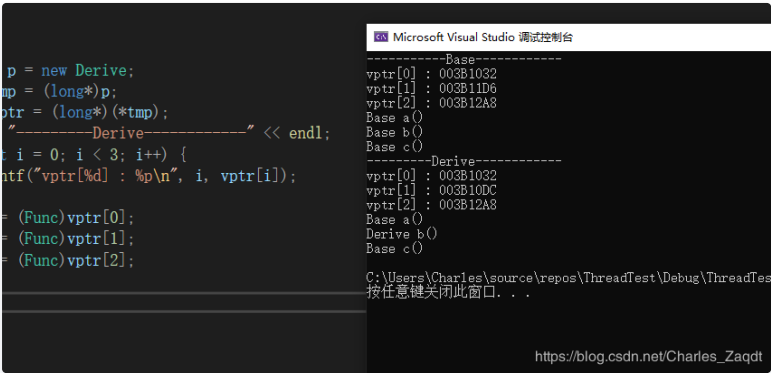
```

1 #include <iostream>
2 #include <stdio.h>
3 using namespace std;
4
5 class Base {
6 public:
7     virtual void a() { cout << "Base a()" << endl; }
8     virtual void b() { cout << "Base b()" << endl; }
9     virtual void c() { cout << "Base c()" << endl; }
10 };
11
12 class Derive : public Base {
13 public:
14     virtual void b() { cout << "Derive b()" << endl; }
15 };
16
17 int main()
18 {
19     typedef void (*Func)();
20     cout << "-----Base-----" << endl;
21     Base* q = new Base;
22     long* tmp1 = (long*)q;
23     long* vptr1 = (long*)(*tmp1);
24     for (int i = 0; i < 3; i++) {
25         printf("vptr[%d] : %p\n", i, vptr1[i]);
26     }
27     Func a = (Func)vptr1[0];
28     Func b = (Func)vptr1[1];

```

```
31     c();
32     c();
33
34     Derive* p = new Derive;
35     long* tmp = (long*)p;
36     long* vptr = (long*)(*tmp);
37     cout << "-----Derive-----" << endl;
38     for (int i = 0; i < 3; i++) {
39         printf("vptr[%d] : %p\n", i, vptr[i]);
40     }
41     Func d = (Func)vptr[0];
42     Func e = (Func)vptr[1];
43     Func f = (Func)vptr[2];
44     d();
45     e();
46     f();
47
48
49     return 0;
50 }
```

运行结果如下：



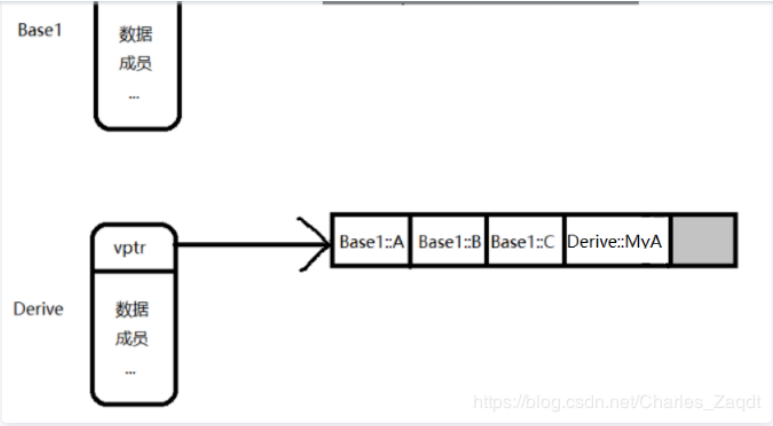
这样就清晰的印证了上述所说的假设，那么虚函数表就获取出来了。

多重继承的虚函数表：

虚函数的引入其实就是为了实现多态(对于多态看到了一篇很不错的博客：[传送门](#))，现在来研究一下多重继承的虚函数表是什么样的，首先我们先来看一下简单的一般继承的代码：

```
1 class Base1 {
2 public:
3     virtual void A() { cout << "Base1 A()" << endl; }
4     virtual void B() { cout << "Base1 B()" << endl; }
5     virtual void C() { cout << "Base1 C()" << endl; }
6 };
7
8 class Derive : public Base1{
9 public:
10     virtual void MyA() { cout << "Derive MyA()" << endl; }
11 };
```

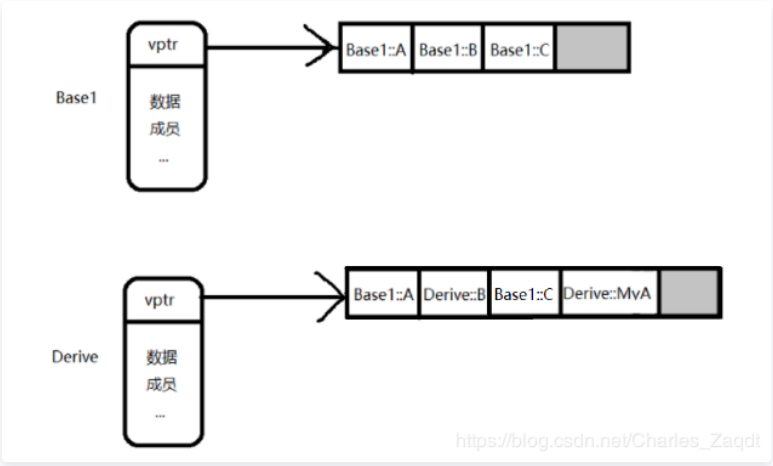
这是一个类继承一个类，这段代码如果我们通过派生类去调用基类的函数，应该结果可想而知，这里就不再演示和赘述了。我们来分析这两个类的虚函数表，对于基类的虚函数表其实和上面所说的虚函数表是一样的，有自己的虚函数指针，并指向自己的虚函数表，重点是在于派生类的虚函数表是什么样子的，它的样子如下图所示：



那么Derive的虚函数表就是继承了Base1的虚函数表，然后自己的虚函数放在后面，因此这个虚函数表的顺序就是基类的虚函数表中的虚函数的顺序+自己的虚函数的顺序。那么我们现在在Derive中再添加一个虚函数，让它覆盖基类中的虚函数，代码如下：

```
1 class Base1 {
2 public:
3     virtual void A() { cout << "Base1 A()" << endl; }
4     virtual void B() { cout << "Base1 B()" << endl; }
5     virtual void C() { cout << "Base1 C()" << endl; }
6 };
7
8 class Derive : public Base1{
9 public:
10     virtual void MyA() { cout << "Derive MyA()" << endl; }
11     virtual void B() { cout << "Derive B()" << endl; }
12 };
```

那么对于这种情况的虚函数表如下图所示：



这个是单继承的情况，然后我们看看多重继承，也就是Derive类继承两个基类，先看一下代码：

```
1 class Base1 {
2 public:
3     virtual void A() { cout << "Base1 A()" << endl; }
4     virtual void B() { cout << "Base1 B()" << endl; }
5     virtual void C() { cout << "Base1 C()" << endl; }
6 };
7
8 class Base2 {
9 public:
10     virtual void D() { cout << "Base2 D()" << endl; }
11     virtual void E() { cout << "Base2 E()" << endl; }
12 };
```

专栏

问答

沙龙

更多

玩转C++
有奖调研

找文章 / 找答案 / 找技术大牛

创作

提问

登录

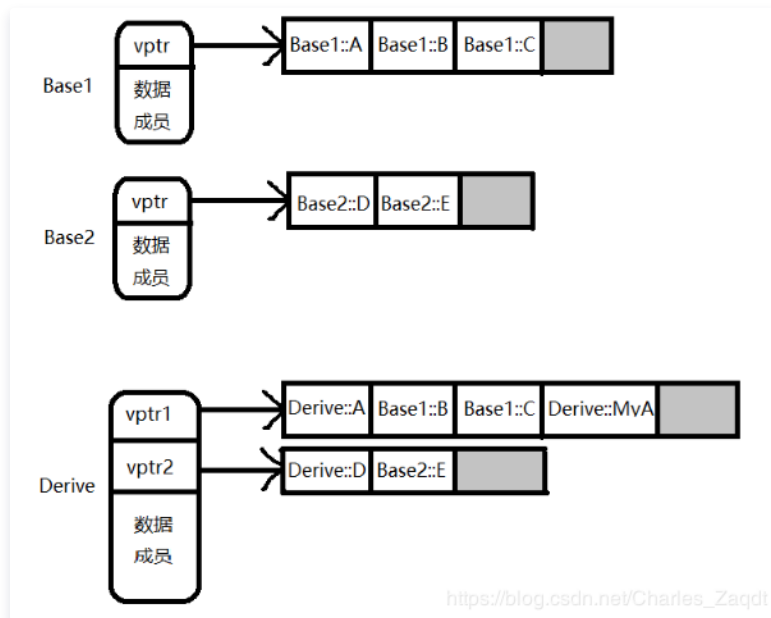
注册

```

16     virtual void A() { cout << "Derive A()" << endl; }
17     virtual void D() { cout << "Derive D()" << endl; }
18     virtual void MyA() { cout << "Derive MyA()" << endl; }
19 };

```

首先我们明确一个概念，对于多重继承的派生类来说，它含有多个虚函数指针，对于上述代码而言，Derive含有两个虚函数指针，所以它不是只有一个虚函数表，然后把所有的虚函数都塞到这个表中，为了印证这一点，我们下面会印证这一点，首先我们先来看看这个多重继承的图示：



由图可以看出，在第一个虚函数表中首先继承了Base1的虚函数表，然后将自己的虚函数放在后面，对于第二个虚函数表中，继承了Base2的虚函数表，由于在Derive类中有一个虚函数D覆盖了Base2的虚函数，所以第一个表中就没有Derive::D的函数地址。那么我们就用代码来实际的验证一下是否会存在两个虚函数指针，以及如果存在两个虚函数表，那么虚函数表是不是这个样子的。来看下面的代码：

```

1  #include <iostream>
2  #include <stdio.h>
3  using namespace std;
4
5  class Base1 {
6  public:
7      virtual void A() { cout << "Base1 A()" << endl; }
8      virtual void B() { cout << "Base1 B()" << endl; }
9      virtual void C() { cout << "Base1 C()" << endl; }
10 };
11
12 class Base2 {
13 public:
14     virtual void D() { cout << "Base2 D()" << endl; }
15     virtual void E() { cout << "Base2 E()" << endl; }
16 };
17
18 class Derive : public Base1, public Base2{
19 public:
20     virtual void A() { cout << "Derive A()" << endl; }
21     virtual void D() { cout << "Derive D()" << endl; }
22     virtual void MyA() { cout << "Derive MyA()" << endl; }
23 };
24
25 int main()
26 {
27     typedef void (*Func)();
28     Derive d;

```

专栏

问答

沙龙

更多

玩转C++
有奖调研

找文章 / 找答案 / 找技术大牛

创作

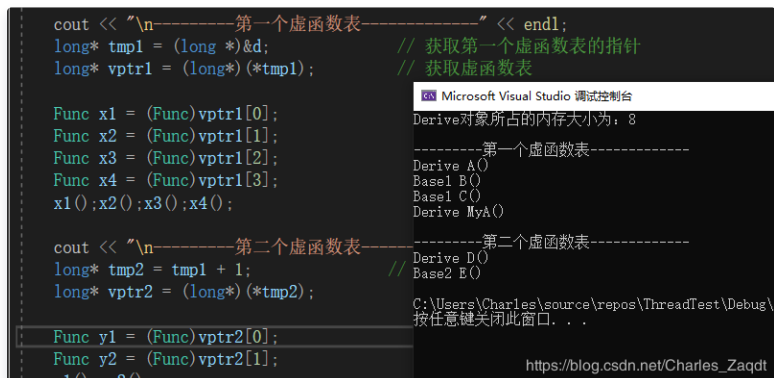
提问

登录

注册

```
32
33     cout << "\n-----第一个虚函数表-----" << endl;
34     long* tmp1 = (long *)&d;           // 获取第一个虚函数表的指针
35     long* vptr1 = (long*)(*tmp1);      // 获取虚函数表
36
37     Func x1 = (Func)vptr1[0];
38     Func x2 = (Func)vptr1[1];
39     Func x3 = (Func)vptr1[2];
40     Func x4 = (Func)vptr1[3];
41     x1();x2();x3();x4();
42
43     cout << "\n-----第二个虚函数表-----" << endl;
44     long* tmp2 = tmp1 + 1;             // 获取第二个虚函数表的指针
45     long* vptr2 = (long*)(*tmp2);
46
47     Func y1 = (Func)vptr2[0];
48     Func y2 = (Func)vptr2[1];
49     y1(); y2();
50
51     return 0;
52 }
```

先看看运行结果，然后再去分析证明：



```
cout << "\n-----第一个虚函数表-----" << endl;
long* tmp1 = (long *)&d;           // 获取第一个虚函数表的指针
long* vptr1 = (long*)(*tmp1);      // 获取虚函数表

Func x1 = (Func)vptr1[0];
Func x2 = (Func)vptr1[1];
Func x3 = (Func)vptr1[2];
Func x4 = (Func)vptr1[3];
x1();x2();x3();x4();

cout << "\n-----第二个虚函数表-----" << endl;
long* tmp2 = tmp1 + 1;             // 获取第二个虚函数表的指针
long* vptr2 = (long*)(*tmp2);

Func y1 = (Func)vptr2[0];
Func y2 = (Func)vptr2[1];
y1(); y2();
```

因为在包含一个虚函数表的时候，含有一个虚函数表指针，所占用的大小为4个字节，那么这里输出了8个字节，就说明Derive对象含有两个虚函数表指针。然后通过获取到了这两个虚函数表，并调用其对应的虚函数，可以发现输出的结果和上面的示例图是相同的，因此就证明了上述所说的结论是正确的。

简单的总结一下：

1. 每一个基类都会有自己的虚函数表，派生类的虚函数表的数量根据继承的基类的数量来定。
2. 派生类的虚函数表的顺序，和继承时的顺序相同。
3. 派生类自己的虚函数放在第一个虚函数表的后面，顺序也是和定义时顺序相同。
4. 对于派生类如果要覆盖父类中的虚函数，那么会在虚函数表中代替其位置。

虚函数指针和虚函数表的创建时机：

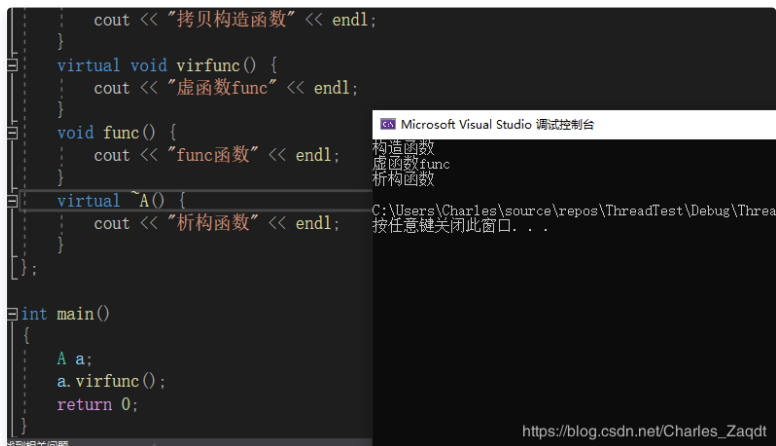
对于虚函数表来说，在编译的过程中编译器就为含有虚函数的类创建了虚函数表，并且编译器会在构造函数中插入一段代码，这段代码用来给虚函数指针赋值。因此虚函数表是在编译的过程中创建。

对于虚函数指针来说，由于虚函数指针是基于对象的，所以对象在实例化的时候，虚函数指针就会创建，所以是在运行时创建。由于在实例化对象的时候会调用到构造函数，所以就会执行虚函数指针的赋值代码，从而将虚函数表的地址赋值给虚函数指针。

虚函数表的深入探索：

```
1 #include <iostream>
2 using namespace std;
3
4 class A{
5 public:
6     int x;
7     A(){
8         memset(this, 0, sizeof(x));    // 将this对象中的
9         cout << "构造函数" << endl;
10    }
11    A(const A& a) {
12        memcpy(this, &a, sizeof(A));    // 直接拷贝内存中的
13        cout << "拷贝构造函数" << endl;
14    }
15    virtual void virfunc() {
16        cout << "虚函数func" << endl;
17    }
18    void func() {
19        cout << "func函数" << endl;
20    }
21    virtual ~A() {
22        cout << "析构函数" << endl;
23    }
24 };
25
26 int main()
27 {
28     A a;
29     a.virfunc();
30     return 0;
31 }
```

在构造函数中用的是memset()函数进行初始化操作，在拷贝构造函数中使用memcpy的方式来拷贝，可能这样的方法效率会更高，其运行结果如下图所示：



```
cout << "拷贝构造函数" << endl;
}
virtual void virfunc() {
    cout << "虚函数func" << endl;
}
void func() {
    cout << "func函数" << endl;
}
virtual ~A() {
    cout << "析构函数" << endl;
}
};

int main()
{
    A a;
    a.virfunc();
    return 0;
}
```

Microsoft Visual Studio 调试控制台

构造函数
虚函数func
析构函数

C:\Users\Charles\source\repos\ThreadTest\Debug\ThreadTest.exe
按任意键关闭此窗口...

https://blog.csdn.net/Charles_Zaqt

可以运行，但是我们要对代码进行分析，前面我们提到了虚函数表是在编译的时候就已经生成好了，那么对于上面的代码中的virfunc来说，它的地址就已经存在于虚函数表中了，又根据前面我们提到的，在实例化对象的时候，编译器会为构造函数中插入一些代码，这些代码用来给虚函数指针进行赋值，那么这些操作都是在我们执行memset之前进行的，因此在执行了这些操作后，调用了memset函数，使得所有内容都清空了，那么虚函数指针就指向了0，那我们为什么还可以调用virfunc函数和析构函数呢？

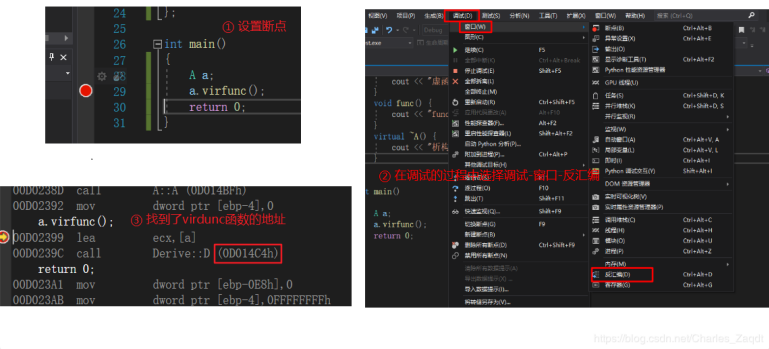
这里就涉及到了静态联编和动态联编的问题，我们先来明确一下静态联编和动态联编的定义：

静态联编：在编译时所进行的这种联编又称静态束定，在编译时就解决了程序中的操作调用与执行该操作代码间的关系。

动态联编：编译程序在编译阶段并不能确切知道将要调用的函数，只有在程序运

联编。

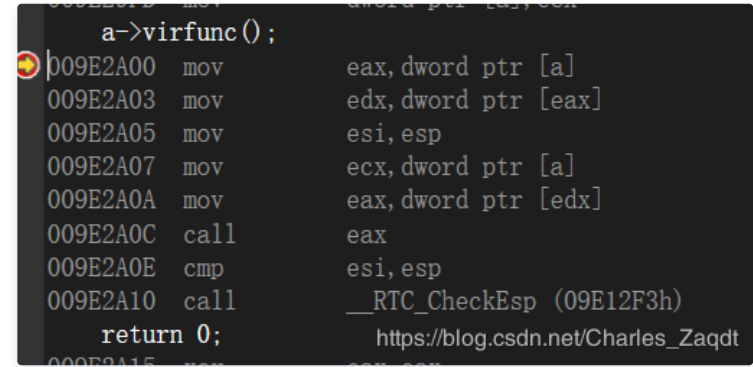
由于我们把虚函数表指针设为了0，所以我们就无法通过前面的方法来获取它，这里我们可以通过反汇编来查看vifunc的地址：



我们发现vifunc函数的地址并不是我们设置的0，那么它就变的和普通的函数没有什么区别了(普通函数采用静态联编，在编译时就绑定了函数的地址)，这显然不是我们想要的虚函数，那么肯定就无法实现多态，对于类的多态性，一定是基于虚函数表的，那么虚函数表的实现一定是动态联编的，因此也不可缺少虚函数指针寻址的过程，那么我们要实现动态联编，就需要用到指针或者引用的形式。如果按上面代码的方式去执行，由于是非指针非引用的形式，所以编译器采用了静态绑定，提前绑定了函数的地址，那么就不存在指针寻址的过程了；如果使用指针或引用的形式，那么由于对象的所属类不能确定，那么编译器就无法采用静态编联，所以就只有通过动态编联的方式，在运行时去绑定调用指针和被调用地址。那么我们把代码改成下面的样子：

```
1 | A *a = new A;
2 | a->vifunc();
```

这个时候我们再运行程序，由于我们将虚函数指针置为0，从而找不到了虚函数的位置，那么程序就会崩溃，然后我们再通过反汇编查看一下，如图所示：



比之前的多了一些汇编指令，其中mov就是寻址的过程，这样就实现了动态绑定，也是根据这一点来实现多态性，这里我只用了一个类来进行展示说明，其实用两个类更好。总之，如果要实现虚函数和多态，就需要通过指针或者引用的方式。

关于虚函数表的东西就是这么多，如果有错误或者遗漏或者有疑问的地方可以在评论区中指出，这篇博客主要是自己学习后的一个总结，对于讲解部分，感觉图片太少了，单纯用文字描述又过于抽象，以后应该要加以改正。

本文参与 腾讯云自媒体分享计划 ，欢迎热爱写作的你一起参与！

本文分享自作者个人站点/博客： https://blog.csdn.net/Charles_Zaqrt 复制

如有侵权，请联系 cloudcommunity@tencent.com 删除。

举报

11

5

登录后参与评论

4 条评论

最新

高赞

- 诺坎普上空的烟花

2022-01-01

作者您好，请问在“虚函数表深入探索”这一段里，类A的构造函数不是将虚函数表的指针指为0了吗？而x没有赋值。为什么说“我们发现vifunc函数的地址并不是我们设置的0”呢，构造函数没有改变虚函数表表项即虚函数的地址吧

0 回复
- 用户9024946 回复 诺坎普上空的烟花

2022-05-09

我替作者再解释一下，说“我们发现vifunc函数的地址并不是我们设置的0”是看图说话，图上显示的是并没有置0，但是我们想的是应该置为0，那么为什么没有置0呢？就是因为这里并不是动态联编，说的更直白一点，A a; a.vifunc();注意这里并不是父类指针或者父类引用指向子类对象，也就是，这里直接实例化对象，并不是用引用或者指针指向对象，所以是静态联编，编译的时候就定死了，不会变的，所以并不是我们想的那样（构造函数把虚函数表指针又置0）

0 回复
- 用户9252141

2021-12-02

好棒，解决了我一大困惑

0 回复
- 用户7235274

2020-09-05

讲解得很棒，谢谢答主

0 回复

相关文章

深入探索虚函数表(详细)

这篇博客可能有一点点长，代码也有一点点多，但是仔细阅读分析完，会对虚函数表有一个深刻的认识。

Ch_Zaqt

深入探索虚函数表(详细)

这篇博客可能有一点点长，代码也有一点点多，但是仔细阅读分析完，会对虚函数表有一个深刻的认识。

从零开始学C++之虚继承和虚函数对C++对...

首先重新回顾一下关于类/对象大小的计算原则： 类大小计算遵循结构体对齐原则 第一个数据成员放在offset为0的位...

s1mba

CVTE2016春季实习校招技术一面回忆（C+...

2016.3.15，参加了CVTE的技术面，很不幸，我和我的两位小伙伴均跪在了一面。先将当日的面试内容汇总如下，供...

Dabelv

C++多态与虚函数表

C++是一门支持面向对象编程(object-oriented Programming)的语言，继承和多态(Polymorphic)是其最重要的特性。 关于C++...

查利鹏

C++单继承、多继承情况下的虚函数表分析

C++的三大特性之一的多态是基于虚函数实现的，而大部分编译器是采用虚函数表来实现虚函数，虚函数表（VTAB）存在于可执行文件的只读数据段中，指向VTAB的虚...

jianghaibobo

学习C/C++的一些书籍和工具

从开始学C语言到现在也有五六年的时间了，也看了不少的好书和烂书，折腾了很多工具(编译器/编辑器圣战)，在这...

查利鹏

图说C++对象模型：对象内存布局详解

Tencent JCoder

每日一问（11） 什么是虚函数

这是餐桌上一个必点菜，大家都知道，根本不是独家秘笈，但是看了无数次文章和book，依然完全被锤，停留 表面上...

程序员小王

C++后台开发必看，这个学习路线必须收藏

在去年结束的秋季招聘中，后台开发或服务器开发的岗位需求一度火热，甚至超过了算法岗。不少同学从诸神黄昏的...

java架构师

浅析C++内存布局

简单总结下C++变量在内存中的布局和可执行文件相关的知识。暂未涉及虚函数，虚函数表，类的继承和多态等C++...

杨永贞

如上示例代码，定义基类BaseClass，BaseClass定义了虚析构函数、虚函数VirtualFunction1、虚函数VirtualFunction2、非...

gaigai

C++类对象所占的内存空间

对于一个什么都没有定义的空类来说，它的大小不是0，而是1，因为实例化对象会获得一个独一无二的地址，也是为了区别该类的不同对象。在深度探索C++对象...

Ch_Zaadt

C++ 虚函数表解析

C++中的虚函数的作用主要是实现了多态的机制。关于多态，简而言之就是用父类型别的指针指向其子类的实例，...

bear_fish

C++多态

在 C++ 程序设计中，多态性是指具有不同功能的函数可以用同一个函数名，这样就可以用一个函数名调用不同内容的...

范中豪

c++ 继承类强制转换时的虚函数表工作原理

本文通过简单例子说明子类之间发生强制转换时虚函数如何调用，旨在对c++继承中的虚函数表的作用机制有更深入的理解。

xiaoxi666

lldb 入坑指北（3） – 打印 c++ 实例的虚函...

打印 c++ 的虚函数表可以快速的帮助我们了解 c++ 父类与子类的 override 关系。但是，lldb 目前却只支持常用的...

酷酷的哀殿

你有一份新的C++书单，请注意查收！

C是C语言的继承，它既可以进行C语言的过程化程序设计，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...

Java技术江湖

更多文章

玩转Cloud有奖调研

找文章 / 找答案 / 找技术大牛

创作

提问

登录

注册

专栏文章

阅读清单

互动问答

技术沙龙

团队主页

热门产品

热门推荐

更多推荐

自媒体分享计划

邀请作者入驻

自荐上首页

技术竞赛

云存储

人脸识别

SSL 证书

数据安全

网站监控

技术周刊

社区标签

开发者手册

开发者实验室

区块链服务

企业云

短信

消息队列

CDN 加速

文字识别

网络加速

视频通话

云点播

云数据库

图像分析

商标注册

域名解析

MySQL 数据库

小程序开发

视频介绍

社区规范

免责声明

联系我们

友情链接



扫码关注腾讯云开发者
领取腾讯云代金券

Copyright © 2013 – 2022 Tencent Cloud. All Rights Reserved. 腾讯云 版权所有 京公网安备 11010802017518 粤B2–20090059–1