不要被C++“自动生成”所蒙骗

2013-01-12 22:40 by Florian, 1652 阅读, 5 评论, 收藏, 编辑

不要被C++“自动生成”所蒙骗

C++对象可以使用两种方式进行创建：构造函数和复制构造函数。假如我们定义了类A，并使用它创建对象。

```
A a,b;
A c=a;
A d(b);
```

对象a和b使用编译器提供的默认构造函数A::A()创建出来，我们称这种创建方式为对象的定义（包含声明的含义）。对象c和d则是使用已有的对象，通过编译器提供的复制构造函数A::A(const A&)创建，我们称这种创建方式为对象的初始化（包含定义和声明的含义）。

可能不少人会把对象的初始化和对象的赋值混淆，比如。

c=d;

这里把对象d赋值给对象c并非创建新的对象，它不会调用任何构造函数。编译器默认提供的赋值运算符重载函数const A&operator=(const A&)为该语句提供支持。

编译器除了提供默认构造函数、复制构造函数和赋值运算符重载函数之外，有可能还为我们提供了析构函数A::~A()，但是这里的析构函数并不是virtual的（相信会有童鞋忘记这一点）。

这些基础的语法对学习过C++的人或许并不陌生，我们自从学习了面向对象C++后，一直都知道编译器为我们提供了这样的便利条件。经过多年的编程实践和体验，我们绝对相信编译器的确为我们做了这些工作，因为我们没有遇到过任何问题。甚至我们脑子中会默认形成一个概念——即使我定义了一个空类（类内什么都没有），编译器依然会“乖乖的”为我们生成上边所说的四个函数。

如果你真的形成了这种观念的话，那么恭喜你，因为你已经将C++基本规则运用的十分熟练了。同时遗憾的是你我都看到了冰山一角，编译器的工作方式远不像我们使用它的那样。读者可能会疑问，难道编译器没有生成这些函数吗？答：要看你类的定义。那么编译器到底如何生成这些函数呢？和我一样又好奇心的人都想一探究竟，而这些内容在《Inside The C++ Object Model》被诠释的比较彻底。笔者也通过“借花献佛”的方式将该书所描述的对象构造的内幕结合个人的理解和大家一起分享。

首先我们从最简单的谈起，编译器为类生成构造函数了吗？如果按照上边描述的例子，只有一个空的类定义的话，我们可以肯定的说——没有。对编译器这样的做法，我们不必感到惊讶。试想一个空的类——没有数据成员，没有成员函数，即使生成了构造函数又能做什么呢？即便是生成了，也只是一个空构造函数而已。

```
A(){}
```

它什么也做不了，也什么都不必做。更“悲剧”，它的出现不仅没有任何积极意义，还会为编译器和程序运行增加完全不必要的函数调用负担。

既然如此，我们让这个类再复杂一点，我们为它增加数据成员和成员函数，比如下边这段代码（我们记它为例子1）。

```
class A
{
public:
        int var;
        void fun(){}
};
```

即便如此，结果还是和上边的一样，不生成构造函数！因为没有任何理由对var初始化，况且编译器也不知道用什么值给它初始化。

```
        A a;
        return 0;
012E1259 33 C0                xor         eax,eax
```

果然，在主函数内定义对象a后，没有任何构造函数被调用。

有人可能会说用0初始化不行吗？这只是我们的“一厢情愿”而已。一个没有初始化的变量

About

范志东（**Florian**），目前为后台开发工程师。本人兴趣广泛，热爱计算机技术，喜欢编程。乐于尝试解决困难问题，对操作系统，编译系统等底层技术有着浓厚的兴趣。希望通过撰写博客分享自己的知识和快乐，与园友们一起进步和提高。如果你与我志同道合，请关注我，让我们共同成长！

本身的值就可以是不确定的，何必要生成一个没有任何意义的初始化为0的语句呢。

编译器到底怎样才能生成构造函数呢？！或许你和我一样有点“抓狂”了。不过现在还不是绝望的时候，因为编译器需要我们给它一个“正当的理由”生成构造函数。有四个正当的理由，让编译器不得不生成构造函数，这里一一介绍。

首先，我们修改一下var的类型。这里假设它不是内置类型int，而是一个定义好的类B。

B var;

修改一下数据成员的类型为自定义类型能影响编译器的抉择吗？答：可能。这要看类B有没有定义构造函数。读者可能有点明白了，是的，如果B没有定义构造函数（和这里的A一个样子），那么编译器仍然没有理由生成构造函数——为B初始化什么呢？反之，B一旦定义了默认构造函数B::B()，即便它是空的，编译器就不得不为A创建默认构造函数了（这里不考虑编译器的深度优化）。因为A的对象需要用B的默认构造函数初始化它自己的成员var，虽然B的构造函数什么也没做。因为编译器不能假定B的构造函数做了什么样的操作（极端一点：万一修改了一个全局变量了呢？），因此编译器有绝对的必要生成A的构造函数，保证B类型的数据成员的构造函数正常执行。

```
    A a;
00A01259 8D 4D FF             lea         ecx,[a]
00A0125C E8 CB FE FF FF       call        A::A (0A0112Ch)
```

转到编译器为A生成的构造函数处，我们发现了B的构造函数被调用的语句（选中行）。

```
A::A:
009C1270 55                   push        ebp
009C1271 8B EC                mov         ebp,esp
009C1273 83 EC 44             sub         esp,44h
009C1276 53                   push        ebx
009C1277 56                   push        esi
009C1278 57                   push        edi
009C1279 89 4D FC             mov         dword ptr [ebp-4],ecx
009C127C 8B 4D FC             mov         ecx,dword ptr [this]
009C127F E8 AD FE FF FF       call        B::B (9C1131h)
009C1284 8B 45 FC             mov         eax,dword ptr [this]
009C1287 5F                   pop         edi
009C1288 5E                   pop         esi
009C1289 5B                   pop         ebx
009C128A 8B E5                mov         esp,ebp
009C128C 5D                   pop         ebp
009C128D C3                   ret
```

当然，如果B提供了构造函数，但不是默认的构造函数，那么必须要程序员介入为var初始化，否则编译器就不客气了——error！

因此，编译器生成默认构造函数的第一个正当理由是——类内数据成员是对象，并且该对象的类提供了一个默认构造函数。

现在，我们回到例子1，这里我们不修改var的类型，而是让A继承于另一个类C。

class A:public C

我们都知道，在C++构造函数初始化语法中，构造函数会先初始化基类C，再初始化自身的数据成员或者对象。因此，这里的问题和对象成员var类似。如果基类C没有提供任何构造函数，那么编译器仍然不提供A的默认构造函数。如果C提供了默认构造函数，结果和前边类似。

```
A::A:
01241270 55                   push        ebp
01241271 8B EC                mov         ebp,esp
01241273 83 EC 44             sub         esp,44h
01241276 53                   push        ebx
01241277 56                   push        esi
01241278 57                   push        edi
01241279 89 4D FC             mov         dword ptr [ebp-4],ecx
0124127C 8B 4D FC             mov         ecx,dword ptr [this]
0124127F E8 B7 FE FF FF       call        C::C (124113Bh)
01241284 8B 45 FC             mov         eax,dword ptr [this]
01241287 5F                   pop         edi
01241288 5E                   pop         esi
01241289 5B                   pop         ebx
0124128A 8B E5                mov         esp,ebp
0124128C 5D                   pop         ebp
0124128D C3                   ret
```

结果不出所料，编译器为A生成了构造函数，并且调用了基类C定义的默认构造函数。同样，若C没有提供默认默认构造函数，而提供了其他构造函数，编译是无法通过的。

这也是编译器生成默认构造函数的第二个正当理由——类的基类提供了默认的构造函数。

我们再次回到例子1，这次我们修改成员函数fun。

virtual void fun(){}

我们把类A的成员函数fun修改为虚函数，再次看看是否产生了默认构造函数。

```
A::A:
00FF12D0 55                   push        ebp
00FF12D1 8B EC                mov         ebp,esp
00FF12D3 83 EC 44             sub         esp,44h
00FF12D6 53                   push        ebx
00FF12D7 56                   push        esi
00FF12D8 57                   push        edi
00FF12D9 89 4D FC             mov         dword ptr [ebp-4],ecx
00FF12DC 8B 45 FC             mov         eax,dword ptr [this]
00FF12DF C7 00 44 59 FF 00    mov         dword ptr [eax],offset A::`vftable' (0FF5944h)
00FF12E5 8B 45 FC             mov         eax,dword ptr [this]
00FF12E8 5F                   pop         edi
00FF12E9 5E                   pop         esi
00FF12EA 5B                   pop         ebx
00FF12EB 8B E5                mov         esp,ebp
00FF12ED 5D                   pop         ebp
00FF12EE C3                   ret
```

这次编译器“毫不客气”的为A生成了默认构造函数，虽然它没有调用任何其他的构造函数！这是什么原因呢？原来，C++为了实现多态机制，需要为类维护一个虚函数表（vftable），而每个该类的对象都保存一个指向该虚函数表的一个指针（一般保存在对象最开始的四个四节处，多态机制的实现这里暂不介绍）。编译器为A生成构造函数，其实不为别的，就为了保证它定义的对象都要正常初始化这个虚函数表的指针（vfptr）！

好了，因此我们得出编译器生成默认构造函数的第三个正当理由——类内定义了虚函数。这里可能还涉及一个更复杂点的情况：类内本身没有定义虚函数，但是继承了基类的虚函数。其实按照上述的原则，我们可以推理如下：基类既然定义了虚函数，那么基类本身就需要生成默认构造函数初始化它本身的虚函数表指针。而基类一旦产生了默认构造函数，派生类就需要产生默认构造函数调用它。同时，如果读者对多态机制了解清除的话，派生类在生成的默认构造函数内还会初始化一次这个虚函数表指针的。

最后，我们再次回到例子1，这次仍然让A继承于C，但是这次C是一个空类——什么都没有，也不会自动生成默认构造函数。但是A继承C的方式要变化一下。

class A:public virtual C

A虚继承于C，这次又有什么不同呢？

```
A::A:
00281280 55                   push        ebp
00281281 8B EC                mov         ebp,esp
00281283 83 EC 44             sub         esp,44h
00281286 53                   push        ebx
00281287 56                   push        esi
00281288 57                   push        edi
00281289 89 4D FC             mov         dword ptr [ebp-4],ecx
0028128C 83 7D 08 00          cmp         dword ptr [ebp+8],0
00281290 74 09                je          A::A+1Bh (28129Bh)
00281292 8B 45 FC             mov         eax,dword ptr [this]
00281295 C7 00 40 59 28 00    mov         dword ptr [eax],offset A::`vbtable' (285940h)
0028129B 8B 45 FC             mov         eax,dword ptr [this]
0028129E 5F                   pop         edi
0028129F 5E                   pop         esi
002812A0 5B                   pop         ebx
002812A1 8B E5                mov         esp,ebp
002812A3 5D                   pop         ebp
002812A4 C2 04 00             ret         4
```

这次编译器也生成了A的构造函数，并且初始化过程和虚函数时有点类似。细心观察下发现，这次构造函数也初始化了一张表——vbtable。了解虚继承机制的读者应该不会陌生，这张表叫虚基类表，它记录了类继承的所有的虚基类子对象在本类定义的对象内的偏移位置（至于虚继承机制的实现，我们以后详细探讨）。为了保证虚继承机制的正确工作，对象必须在初始化阶段维护一个指向该表的一个指针，称为虚表指针（vbptr）。编译器因为它提供A的默认构造函数的理由和虚函数时类似。

这样，我们得出编译器生成默认构造函数的第四个正当理由——类使用了虚继承。

到这里，我们把编译器为类生成默认构造函数的正当理由阐述完毕，相信大家应该对构造函数的生成时机有了一个大致的认识。这四种“正当理由”其实是编译器不得不为类生成默认构造函数的理由，《Inside The C++ Object Model》里称这种理由为nontrival的（候sir翻译的很别扭，所以怎么翻译随你啦）。除了这四种情况外，编译器称为trival的，也就是没有必要为类生成默认构造函数。这里讨论的构造函数生成准则的内容是写进C++Standard的，如此看来标准就是“贴合正常思维”的一套准则（简单YY一下），其实本就是这样，编译器不应该为了一致化做一些没有必要的工作。

通过对默认构造函数的讨论，相信大家对复制构造函数、赋值运算符重载函数、析构函数的生成时机应该可以自动扩展了。没错，它们遵循着一个最根本的原则：只有编译器不得不为这个类生成函数的时候（**nontrival**），编译器才会真正的生成它。

因此，正如标题所说，我们不要被C++语法中所描述的那些条条框框所“蒙骗”了。的确，相信这些生成规则不会对我们的编程带来多大的影响（不会产生错误），但是只有了解它们的背

后操作，我们才知道编译器究竟为我们做了什么，我们才知道如何使用C++才能让它变得更有效率——比如消除不必要的构造和虚拟机制等（如果可以的话）。相信本文对C++自动生成的内容的描述让不少人认清对象构造函数产生的前因后果，希望本文对你有所帮助。

作者：Florian

Florian
关注 - 15
粉丝 - 364

2 0

荣誉：推荐博客

+加关注

(请您对文章做出评价)

« 上一篇：组合模式（Composition）
» 下一篇：对象的传值与返回

分类: C++

标签: C++, trival, 自动生成

ADD YOUR COMMENT

#1楼 蓝云在天
2013-01-13 11:41

非常细致！楼主在C++方面工作多久了？

支持(0) 反对(0)

#2楼[楼主] Florian
2013-01-13 13:18

@蓝云在天
从本科开始学习C++，算是有6个年头了吧！

支持(0) 反对(0)

#3楼 roxma
2013-01-13 13:34

其实总结起来就是安全和效率。

支持(0) 反对(0)

#4楼 wyjiuye
2013-01-15 21:29

总结的细致。

支持(0) 反对(0)

#5楼 akima
2013-01-25 04:26

mark 一个

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:
· 你不知道的连续创业者：成功是不正常的
· 波罗蜜完成3000万美元B轮融资：百度参投
· 搜狗起诉百度输入法专利侵权 索赔8千万
· 小伙被骗13次总计15万元：骗子主动拉黑都没用
· “iPod之父” 透露与妻子离开苹果公司的原因
» 更多新闻...

最新知识库文章:
· 什么时候应该避免写代码注释？
· 持续集成是什么？
· 人，技术与流程
· HTTPS背后的加密算法
· 下一代云计算模式：Docker正掀起个性化商业革命
» 更多知识库文章...