嵌入式C++开发

常见知识点

整理by Yang，Gavin

2016-12-12

# 1、哪种情况必须用到初始化成员列表

必须用到初始化成员列表的四种情况：

1）初始化一个reference成员；  
2）初始化一个const成员；  
3）调用一个基类的[构造函数](https://www.baidu.com/s?wd=%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkmyRsnyNWnHfsuhcznW7h0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHbdrHbdrHbsrj0LPHf3rH0zn0)，而该函数有一组参数；  
4）调用一个数据成员对象的[构造函数](https://www.baidu.com/s?wd=%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkmyRsnyNWnHfsuhcznW7h0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHbdrHbdrHbsrj0LPHf3rH0zn0)，而该函数有一组参数。

举例说明：

|  |  |
| --- | --- |
|  | #include <iostream>  using namespace std;    class Base  {  public:      Base(const string &str = "", int i = 0) : Bstr(str), \_i(i) // 使用const引用避免复制,      // 如果只使用const则无法使用字面常量"DerivedStr"为str赋值      {          cout << "Base Constructor" << " Bstr = " << Bstr << ", \_i = " << \_i << endl;      }        string Bstr;      int \_i;  };  class Derived : public Base  {  public:        // 调用基类构造函数，而它拥有一组参数时，要使用成员初始化列表      Derived() : Base("DerivedStr", 200)  // 这个是正确的      {          //Base::Bstr = "DerivedStr"; // 基类构造函数再次之前调用，这里赋值没有用。          //Base::\_i = 200;          cout << "Derived Constructor" << endl;      }        string Dstr;  };  int main()  {      Derived d;      return 0;  } |

# 2、说一下C++的虚函数

通过虚函数实现了C++程序在运行时的多态。每个带有虚函数的类对象都有一个指向虚函数表的指针。这个指针位于类对象内存空间的顶部。父类的虚函数被子类重写后，在虚函数表中，子类的虚函数地址会覆盖父类的这个虚函数地址。当在使用父类指针去执行这个虚函数时，就会执行子类的虚函数。这就实现了多态的性质。**虚函数表是存放在静态存储区里面**。

# 3、什么时候应必须使用dynamic\_cast？什么时候dynamic\_cast可以使用[static\_cast](http://baike.baidu.com/view/1745207.htm)代替？

1、对应单继承，如果父类的指针指向子类对象，用dynamic\_cast和static\_cast效果一样；

2、如果父类指针不是真的执行子类对象，用dynamic\_cast则返回NULL，能够更早的禁止error的发生。如果用static\_cast虽然返回的不是NULL，但在运行时可能抛出异常。

3、对应多重继承，如果父类Base指针真的指向子类对象，使用dynamic\_cast和static\_cast都可以转换为子类对象指针。但如果要转化为父类的兄弟类Base1，必须使用dynamic\_cast,使用static\_cast不能编译。

4、对应不是真的指向子类对象，要想转化为子类指针或兄弟类对象指针，用dynamic\_cast则返回NULL。如果用static\_cast虽然返回的不是NULL，但在运行时可能抛出异常。

# 4、qobject\_cast与dynamic区别

T qobject\_cast(QObject \*obj)

类T必须继承自QObject类，并且使用Q\_OBJECT对象。qobject\_cast类似于dynamic\_cast，但它不需要RTTI的支持。qobject\_cast依赖QMetaObject信息来判断。

# 5、static\_cast, dynamic\_cast, const\_cast, reinterpret\_cast

dynamic\_cast: 通常在基类和派生类之间转换时使用,run-time cast  
const\_cast: 主要针对const和volatile的转换.  
static\_cast: 一般的转换，no run-time check.通常，如果你不知道该用哪个，就用这个。  
reinterpret\_cast: 用于进行没有任何关联之间的转换，比如一个字符指针转换为一个整形数。

# [C++ 虚函数表解析](http://blog.csdn.net/haoel/article/details/1948051)

## 前言

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，然后通过父类的指针调用实际子类的成员函数。这种技术可以让父类的指针有“多种形态”，这是一种泛型技术。所谓泛型技术，说白了就是试图使用不变的代码来实现可变的[**算法**](http://lib.csdn.net/base/datastructure)。比如：模板技术，RTTI技术，虚函数技术，要么是试图做到在编译时决议，要么试图做到运行时决议。

关于虚函数的使用方法，我在这里不做过多的阐述。大家可以看看相关的C++的书籍。在这篇文章中，我只想从虚函数的实现机制上面为大家 一个清晰的剖析。

当然，相同的文章在网上也出现过一些了，但我总感觉这些文章不是很容易阅读，大段大段的代码，没有图片，没有详细的说明，没有比较，没有举一反三。不利于学习和阅读，所以这是我想写下这篇文章的原因。也希望大家多给我提意见。

言归正传，让我们一起进入虚函数的世界。

## 虚函数表

对C++ 了解的人都应该知道虚函数（Virtual Function）是通过一张虚函数表（Virtual Table）来实现的。简称为V-Table。在这个表中，主是要一个类的虚函数的地址表，这张表解决了继承、覆盖的问题，保证其容真实反应实际的函数。这样，在有虚函数的类的实例中这个表被分配在了这个实例的内存中，所以，当我们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由为重要了，它就像一个地图一样，指明了实际所应该调用的函数。

这里我们着重看一下这张虚函数表。C++的编译器应该是保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证取到虚函数表的有最高的性能——如果有多层继承或是多重继承的情况下）。 这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。

听我扯了那么多，我可以感觉出来你现在可能比以前更加晕头转向了。 没关系，下面就是实际的例子，相信聪明的你一看就明白了。

假设我们有这样的一个类：

class Base {

     public:

            virtual void f() { cout << "Base::f" << endl; }

            virtual void g() { cout << "Base::g" << endl; }

            virtual void h() { cout << "Base::h" << endl; }

};

按照上面的说法，我们可以通过Base的实例来得到虚函数表。 下面是实际例程：

          typedef void(\*Fun)(void);

            Base b;

            Fun pFun = NULL;

            cout << "虚函数表地址：" << (int\*)(&b) << endl;

            cout << "虚函数表 — 第一个函数地址：" << (int\*)\*(int\*)(&b) << endl;

            // Invoke the first virtual function

            pFun = (Fun)\*((int\*)\*(int\*)(&b));

            pFun();

实际运行经果如下：(Windows XP+VS2003,  [**Linux**](http://lib.csdn.net/base/linux) 2.6.22 + GCC 4.1.3)

虚函数表地址：0012FED4

虚函数表 — 第一个函数地址：0044F148

Base::f

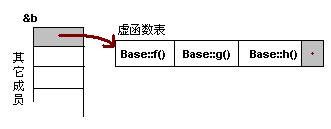
通过这个示例，我们可以看到，我们可以通过强行把&b转成int \*，取得虚函数表的地址，然后，再次取址就可以得到第一个虚函数的地址了，也就是Base::f()，这在上面的程序中得到了验证（把int\* 强制转成了函数指针）。通过这个示例，我们就可以知道如果要调用Base::g()和Base::h()，其代码如下：

            (Fun)\*((int\*)\*(int\*)(&b)+0);  // Base::f()

            (Fun)\*((int\*)\*(int\*)(&b)+1);  // Base::g()

            (Fun)\*((int\*)\*(int\*)(&b)+2);  // Base::h()

这个时候你应该懂了吧。什么？还是有点晕。也是，这样的代码看着太乱了。没问题，让我画个图解释一下。如下所示：

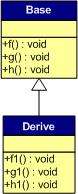


注意：在上面这个图中，我在虚函数表的最后多加了一个结点，这是虚函数表的结束结点，就像字符串的结束符“/0”一样，其标志了虚函数表的结束。这个结束标志的值在不同的编译器下是不同的。在WinXP+VS2003下，这个值是NULL。而在Ubuntu 7.10 + Linux 2.6.22 + GCC 4.1.3下，这个值是如果1，表示还有下一个虚函数表，如果值是0，表示是最后一个虚函数表。

下面，我将分别说明“无覆盖”和“有覆盖”时的虚函数表的样子。没有覆盖父类的虚函数是毫无意义的。我之所以要讲述没有覆盖的情况，主要目的是为了给一个对比。在比较之下，我们可以更加清楚地知道其内部的具体实现。

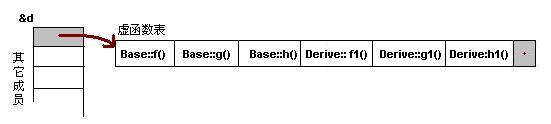
## 一般继承（无虚函数覆盖）

 下面，再让我们来看看继承时的虚函数表是什么样的。假设有如下所示的一个继承关系：



请注意，在这个继承关系中，子类没有重载任何父类的函数。那么，在派生类的实例中，其虚函数表如下所示：

对于实例：Derive d; 的虚函数表如下：



我们可以看到下面几点：

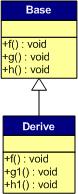
1）虚函数按照其声明顺序放于表中。

2）父类的虚函数在子类的虚函数前面。

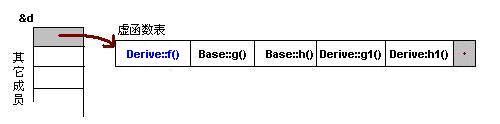
 我相信聪明的你一定可以参考前面的那个程序，来编写一段程序来验证。

## 一般继承（有虚函数覆盖）

覆盖父类的虚函数是很显然的事情，不然，虚函数就变得毫无意义。下面，我们来看一下，如果子类中有虚函数重载了父类的虚函数，会是一个什么样子？假设，我们有下面这样的一个继承关系。



为了让大家看到被继承过后的效果，在这个类的设计中，我只覆盖了父类的一个函数：f()。那么，对于派生类的实例，其虚函数表会是下面的一个样子：



我们从表中可以看到下面几点，

1）覆盖的f()函数被放到了虚表中原来父类虚函数的位置。

2）没有被覆盖的函数依旧。

这样，我们就可以看到对于下面这样的程序，

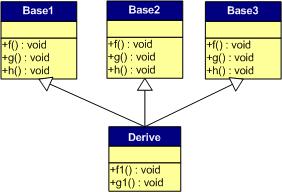
            Base \*b = new Derive();

            b->f();

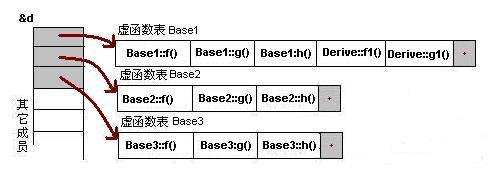
由b所指的内存中的虚函数表的f()的位置已经被Derive::f()函数地址所取代，于是在实际调用发生时，是Derive::f()被调用了。这就实现了多态。

## 多重继承（无虚函数覆盖）

 下面，再让我们来看看多重继承中的情况，假设有下面这样一个类的继承关系。注意：子类并没有覆盖父类的函数。



对于子类实例中的虚函数表，是下面这个样子：



我们可以看到：

1）  每个父类都有自己的虚表。

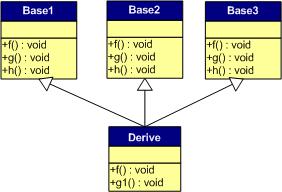
2）  子类的成员函数被放到了第一个父类的表中。（所谓的第一个父类是按照声明顺序来判断的）

这样做就是为了解决不同的父类类型的指针指向同一个子类实例，而能够调用到实际的函数。

## 多重继承（有虚函数覆盖）

 下面我们再来看看，如果发生虚函数覆盖的情况。

 下图中，我们在子类中覆盖了父类的f()函数。



下面是对于子类实例中的虚函数表的图：



我们可以看见，三个父类虚函数表中的f()的位置被替换成了子类的函数指针。这样，我们就可以任一静态类型的父类来指向子类，并调用子类的f()了。如：

            Derive d;

            Base1 \*b1 = &d;

            Base2 \*b2 = &d;

            Base3 \*b3 = &d;

            b1->f(); //Derive::f()

            b2->f(); //Derive::f()

            b3->f(); //Derive::f()

            b1->g(); //Base1::g()

            b2->g(); //Base2::g()

            b3->g(); //Base3::g()

## 安全性

每次写C++的文章，总免不了要批判一下C++。这篇文章也不例外。通过上面的讲述，相信我们对虚函数表有一个比较细致的了解了。水可载舟，亦可覆舟。下面，让我们来看看我们可以用虚函数表来干点什么坏事吧。

**一、通过父类型的指针访问子类自己的虚函数**

我们知道，子类没有重载父类的虚函数是一件毫无意义的事情。因为多态也是要基于函数重载的。虽然在上面的图中我们可以看到Base1的虚表中有Derive的虚函数，但我们根本不可能使用下面的语句来调用子类的自有虚函数：

          Base1 \*b1 = new Derive();

            b1->f1();  //编译出错

任何妄图使用父类指针想调用子类中的**未覆盖父类的成员函数**的行为都会被编译器视为非法，所以，这样的程序根本无法编译通过。但在运行时，我们可以通过指针的方式访问虚函数表来达到违反C++语义的行为。（关于这方面的尝试，通过阅读后面附录的代码，相信你可以做到这一点）

**二、访问non-public的虚函数**

另外，如果父类的虚函数是private或是protected的，但这些非public的虚函数同样会存在于虚函数表中，所以，我们同样可以使用访问虚函数表的方式来访问这些non-public的虚函数，这是很容易做到的。

如：

class Base {

    private:

            virtual void f() { cout << "Base::f" << endl; }

};

class Derive : public Base{

};

typedef void(\*Fun)(void);

void main() {

    Derive d;

    Fun  pFun = (Fun)\*((int\*)\*(int\*)(&d)+0);

    pFun();

}

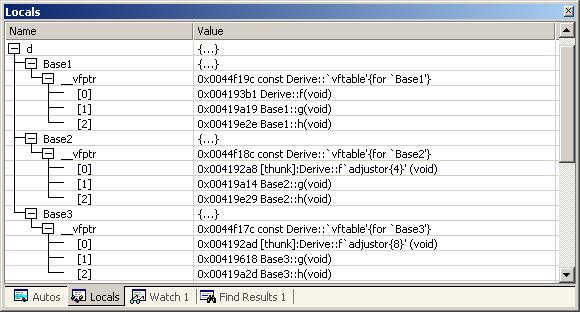
## 结束语

C++这门语言是一门Magic的语言，对于程序员来说，我们似乎永远摸不清楚这门语言背着我们在干了什么。需要熟悉这门语言，我们就必需要了解C++里面的那些东西，需要去了解C++中那些危险的东西。不然，这是一种搬起石头砸自己脚的编程语言。

在文章束之前还是介绍一下自己吧。我从事软件研发有十个年头了，目前是软件开发技术主管，技术方面，主攻Unix/C/C++，比较喜欢网络上的技术，比如分布式计算，网格计算，P2P，Ajax等一切和互联网相关的东西。管理方面比较擅长于团队建设，技术趋势分析，项目管理。欢迎大家和我交流，我的MSN和Email是：[haoel@hotmail.com](mailto:haoel@hotmail.com)

## 附录一：VC中查看虚函数表

我们可以在VC的IDE环境中的Debug状态下展开类的实例就可以看到虚函数表了（并不是很完整的）



## 附录 二：例程

下面是一个关于多重继承的虚函数表访问的例程：

#include <iostream>

using namespace std;

class Base1 {

public:

            virtual void f() { cout << "Base1::f" << endl; }

            virtual void g() { cout << "Base1::g" << endl; }

            virtual void h() { cout << "Base1::h" << endl; }

};

class Base2 {

public:

            virtual void f() { cout << "Base2::f" << endl; }

            virtual void g() { cout << "Base2::g" << endl; }

            virtual void h() { cout << "Base2::h" << endl; }

};

class Base3 {

public:

            virtual void f() { cout << "Base3::f" << endl; }

            virtual void g() { cout << "Base3::g" << endl; }

            virtual void h() { cout << "Base3::h" << endl; }

};

class Derive : public Base1, public Base2, public Base3 {

public:

            virtual void f() { cout << "Derive::f" << endl; }

            virtual void g1() { cout << "Derive::g1" << endl; }

};

typedef void(\*Fun)(void);

int main()

{

            Fun pFun = NULL;

            Derive d;

            int\*\* pVtab = (int\*\*)&d;

**//Base1's vtable**

            //pFun = (Fun)\*((int\*)\*(int\*)((int\*)&d+0)+0);

            pFun = (Fun)pVtab[0][0];

            pFun();

            //pFun = (Fun)\*((int\*)\*(int\*)((int\*)&d+0)+1);

            pFun = (Fun)pVtab[0][1];

            pFun();

            //pFun = (Fun)\*((int\*)\*(int\*)((int\*)&d+0)+2);

            pFun = (Fun)pVtab[0][2];

            pFun();

            //Derive's vtable

            //pFun = (Fun)\*((int\*)\*(int\*)((int\*)&d+0)+3);

            pFun = (Fun)pVtab[0][3];

            pFun();

            //The tail of the vtable

            pFun = (Fun)pVtab[0][4];

            cout<<pFun<<endl;

**//Base2's vtable**

            //pFun = (Fun)\*((int\*)\*(int\*)((int\*)&d+1)+0);

            pFun = (Fun)pVtab[1][0];

            pFun();

            //pFun = (Fun)\*((int\*)\*(int\*)((int\*)&d+1)+1);

            pFun = (Fun)pVtab[1][1];

            pFun();

            pFun = (Fun)pVtab[1][2];

            pFun();

            //The tail of the vtable

            pFun = (Fun)pVtab[1][3];

            cout<<pFun<<endl;

**//Base3's vtable**

            //pFun = (Fun)\*((int\*)\*(int\*)((int\*)&d+1)+0);

            pFun = (Fun)pVtab[2][0];

            pFun();

            //pFun = (Fun)\*((int\*)\*(int\*)((int\*)&d+1)+1);

            pFun = (Fun)pVtab[2][1];

            pFun();

            pFun = (Fun)pVtab[2][2];

            pFun();

            //The tail of the vtable

            pFun = (Fun)pVtab[2][3];

            cout<<pFun<<endl;

            return 0;

}

# C++ 对象的内存布局(上)

## 前言

07年12月，我写了一篇《[C++虚函数表解析](http://blog.csdn.net/haoel/archive/2007/12/18/1948051.aspx)》的文章，引起了大家的兴趣。有很多朋友对我的文章留了言，有鼓励我的，有批评我的，还有很多问问题的。我在这里一并对大家的留言表示感谢。这也是我为什么再写一篇续言的原因。因为，在上一篇文章中，我用了的示例都是非常简单的，主要是为了说明一些机理上的问题，也是为了图一些表达上方便和简单。不想，这篇文章成为了打开C++对象模型内存布局的一个引子，引发了大家对C++对象的更深层次的讨论。当然，我之前的文章还有很多方面没有涉及，从我个人感觉下来，在谈论虚函数表里，至少有以下这些内容没有涉及：

1）有成员变量的情况。

2）有重复继承的情况。

3）有虚拟继承的情况。

4）有钻石型虚拟继承的情况。

这些都是我本篇文章需要向大家说明的东西。所以，这篇文章将会是《[C++虚函数表解析](http://blog.csdn.net/haoel/archive/2007/12/18/1948051.aspx)》的一个续篇，也是一篇高级进阶的文章。我希望大家在读这篇文章之前对C++有一定的基础和了解，并能先读我的上一篇文章。因为这篇文章的深度可能会比较深，而且会比较杂乱，我希望你在读本篇文章时不会有大脑思维紊乱导致大脑死机的情况。;-)

## 对象的影响因素

简而言之，我们一个类可能会有如下的影响因素：

1）成员变量

2）虚函数（产生虚函数表）

3）单一继承（只继承于一个类）

4）多重继承（继承多个类）

5）重复继承（继承的多个父类中其父类有相同的超类）

6）虚拟继承（使用virtual方式继承，为了保证继承后父类的内存布局只会存在一份）

上述的东西通常是C++这门语言在语义方面对对象内部的影响因素，当然，还会有编译器的影响（比如优化），还有字节对齐的影响。在这里我们都不讨论，我们只讨论C++语言上的影响。

本篇文章着重讨论下述几个情况下的C++对象的内存布局情况。

1）**单一的一般继承**（带成员变量、虚函数、虚函数覆盖）

2）**单一的虚拟继承**（带成员变量、虚函数、虚函数覆盖）

3）**多重继承**（带成员变量、虚函数、虚函数覆盖）

4）**重复多重继承**（带成员变量、虚函数、虚函数覆盖）

5）**钻石型的虚拟多重继承**（带成员变量、虚函数、虚函数覆盖）

我们的目标就是，让事情越来越复杂。

## 知识复习

 我们简单地复习一下，我们可以通过对象的地址来取得虚函数表的地址，如：

          typedef void(\*Fun)(void);

            Base b;

            Fun pFun = NULL;

            cout << "虚函数表地址：" << (int\*)(&b) << endl;

            cout << "虚函数表 — 第一个函数地址：" << (int\*)\*(int\*)(&b) << endl;

            // Invoke the first virtual function

            pFun = (Fun)\*((int\*)\*(int\*)(&b));

            pFun();

我们同样可以用这种方式来取得整个对象实例的内存布局。因为这些东西在内存中都是连续分布的，我们只需要使用适当的地址偏移量，我们就可以获得整个内存对象的布局。

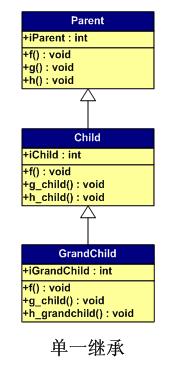
本篇文章中的例程或内存布局主要使用如下编译器和系统：

1）Windows XP 和 VC++ 2003

2）Cygwin 和 G++ 3.4.4

## 单一的一般继承

下面，我们假设有如下所示的一个继承关系：



请注意，在这个继承关系中，父类，子类，孙子类都有自己的一个成员变量。而子类覆盖了父类的f()方法，孙子类覆盖了子类的g\_child()及其超类的f()。

我们的源程序如下所示：

class Parent {

public:

    int iparent;

    Parent ():iparent (10) {}

    virtual void f() { cout << " Parent::f()" << endl; }

    virtual void g() { cout << " Parent::g()" << endl; }

    virtual void h() { cout << " Parent::h()" << endl; }

};

class Child : public Parent {

public:

    int ichild;

    Child():ichild(100) {}

    virtual void f() { cout << "Child::f()" << endl; }

    virtual void g\_child() { cout << "Child::g\_child()" << endl; }

    virtual void h\_child() { cout << "Child::h\_child()" << endl; }

};

class GrandChild : public Child{

public:

    int igrandchild;

    GrandChild():igrandchild(1000) {}

    virtual void f() { cout << "GrandChild::f()" << endl; }

    virtual void g\_child() { cout << "GrandChild::g\_child()" << endl; }

    virtual void h\_grandchild() { cout << "GrandChild::h\_grandchild()" << endl; }

};

我们使用以下程序作为[**测试**](http://lib.csdn.net/base/softwaretest)程序：（下面程序中，我使用了一个int\*\* pVtab 来作为遍历对象内存布局的指针，这样，我就可以方便地像使用数组一样来遍历所有的成员包括其虚函数表了，在后面的程序中，我也是用这样的方法的，请不必感到奇怪，）

    typedef void(\*Fun)(void);

    GrandChild gc;

    int\*\* pVtab = (int\*\*)&gc;

    cout << "[0] GrandChild::\_vptr->" << endl;

    for (int i=0; (Fun)pVtab[0][i]!=NULL; i++){

                pFun = (Fun)pVtab[0][i];

                cout << "    ["<<i<<"] ";

                pFun();

    }

    cout << "[1] Parent.iparent = " << (int)pVtab[1] << endl;

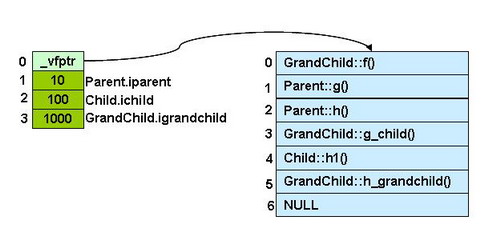
    cout << "[2] Child.ichild = " << (int)pVtab[2] << endl;

    cout << "[3] GrandChild.igrandchild = " << (int)pVtab[3] << endl;

其运行结果如下所示：（在VC++ 2003和G++ 3.4.4下）

|  |
| --- |
| **[0] GrandChild::\_vptr->**  **[0] GrandChild::f()**  **[1] Parent::g()**  **[2] Parent::h()**  **[3] GrandChild::g\_child()**  **[4] Child::h1()**  **[5] GrandChild::h\_grandchild()**  **[1] Parent.iparent = 10**  **[2] Child.ichild = 100**  **[3] GrandChild.igrandchild = 1000** |

使用图片表示如下：



可见以下几个方面：

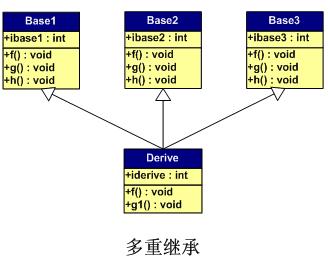
1）虚函数表在最前面的位置。

2）成员变量根据其继承和声明顺序依次放在后面。

3）在单一的继承中，被overwrite的虚函数在虚函数表中得到了更新。

## 多重继承

下面，再让我们来看看多重继承中的情况，假设有下面这样一个类的继承关系。注意：子类只overwrite了父类的f()函数，而还有一个是自己的函数（我们这样做的目的是为了用g1()作为一个标记来标明子类的虚函数表）。而且每个类中都有一个自己的成员变量：



我们的类继承的源代码如下所示：父类的成员初始为10，20，30，子类的为100

class Base1 {

public:

    int ibase1;

    Base1():ibase1(10) {}

    virtual void f() { cout << "Base1::f()" << endl; }

    virtual void g() { cout << "Base1::g()" << endl; }

    virtual void h() { cout << "Base1::h()" << endl; }

};

class Base2 {

public:

    int ibase2;

    Base2():ibase2(20) {}

    virtual void f() { cout << "Base2::f()" << endl; }

    virtual void g() { cout << "Base2::g()" << endl; }

    virtual void h() { cout << "Base2::h()" << endl; }

};

class Base3 {

public:

    int ibase3;

    Base3():ibase3(30) {}

    virtual void f() { cout << "Base3::f()" << endl; }

    virtual void g() { cout << "Base3::g()" << endl; }

    virtual void h() { cout << "Base3::h()" << endl; }

};

class Derive : public Base1, public Base2, public Base3 {

public:

    int iderive;

    Derive():iderive(100) {}

    virtual void f() { cout << "Derive::f()" << endl; }

    virtual void g1() { cout << "Derive::g1()" << endl; }

};

我们通过下面的程序来查看子类实例的内存布局：下面程序中，注意我使用了一个s变量，其中用到了sizof(Base)来找下一个类的偏移量。（因为我声明的是int成员，所以是4个字节，所以没有对齐问题。关于内存的对齐问题，大家可以自行试验，我在这里就不多说了）

         typedef void(\*Fun)(void);

            Derive d;

                int\*\* pVtab = (int\*\*)&d;

                cout << "[0] Base1::\_vptr->" << endl;

                pFun = (Fun)pVtab[0][0];

                cout << "     [0] "; pFun();

                pFun = (Fun)pVtab[0][1];

                cout << "     [1] ";pFun();

                pFun = (Fun)pVtab[0][2];

                cout << "     [2] ";pFun();

                pFun = (Fun)pVtab[0][3];

                cout << "     [3] "; pFun();

                pFun = (Fun)pVtab[0][4];

                cout << "     [4] "; cout<<pFun<<endl;

                cout << "[1] Base1.ibase1 = " << (int)pVtab[1] << endl;

**int s = sizeof(Base1)/4;**

                cout << "[" << s << "] Base2::\_vptr->"<<endl;

                pFun = (Fun)pVtab[s][0];

                cout << "     [0] "; pFun();

                Fun = (Fun)pVtab[s][1];

                cout << "     [1] "; pFun();

                pFun = (Fun)pVtab[s][2];

                cout << "     [2] "; pFun();

                pFun = (Fun)pVtab[s][3];

                out << "     [3] ";

                cout<<pFun<<endl;

                cout << "["<< s+1 <<"] Base2.ibase2 = " << (int)pVtab[s+1] << endl;

**s = s + sizeof(Base2)/4;**

                cout << "[" << s << "] Base3::\_vptr->"<<endl;

                pFun = (Fun)pVtab[s][0];

                cout << "     [0] "; pFun();

                pFun = (Fun)pVtab[s][1];

                cout << "     [1] "; pFun();

                pFun = (Fun)pVtab[s][2];

                cout << "     [2] "; pFun();

                pFun = (Fun)pVtab[s][3];

                 cout << "     [3] ";

                cout<<pFun<<endl;

                s++;

                cout << "["<< s <<"] Base3.ibase3 = " << (int)pVtab[s] << endl;

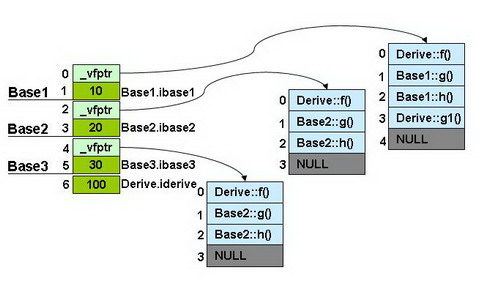
                s++;

                cout << "["<< s <<"] Derive.iderive = " << (int)pVtab[s] << endl;

其运行结果如下所示：（在VC++ 2003和G++ 3.4.4下）

|  |
| --- |
| **[0] Base1::\_vptr->**  **[0] Derive::f()**  **[1] Base1::g()**  **[2] Base1::h()**  **[3] Driver::g1()**  **[4] 00000000       注意：在GCC下，这里是1**  **[1] Base1.ibase1 = 10**  **[2] Base2::\_vptr->**  **[0] Derive::f()**  **[1] Base2::g()**  **[2] Base2::h()**  **[3] 00000000       注意：在GCC下，这里是1**  **[3] Base2.ibase2 = 20**  **[4] Base3::\_vptr->**  **[0] Derive::f()**  **[1] Base3::g()**  **[2] Base3::h()**  **[3] 00000000**  **[5] Base3.ibase3 = 30**  **[6] Derive.iderive = 100** |

使用图片表示是下面这个样子：



我们可以看到：

1）  每个父类都有自己的虚表。

2）  子类的成员函数被放到了第一个父类的表中。

3）  内存布局中，其父类布局依次按声明顺序排列。

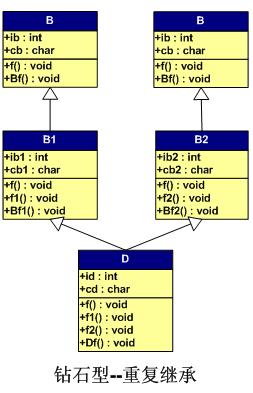
4）  每个父类的虚表中的f()函数都被overwrite成了子类的f()。这样做就是为了解决不同的父类类型的指针指向同一个子类实例，而能够调用到实际的函数。

# C++ 对象的内存布局(下)

## 重复继承

下面我们再来看看，发生重复继承的情况。所谓重复继承，也就是某个基类被间接地重复继承了多次。

下图是一个继承图，我们重载了父类的f()函数。



其类继承的源代码如下所示。其中，每个类都有两个变量，一个是整形（4字节），一个是字符（1字节），而且还有自己的虚函数，自己overwrite父类的虚函数。如子类D中，f()覆盖了超类的函数， f1()和f2() 覆盖了其父类的虚函数，Df()为自己的虚函数。

class B

{

    public:

        int ib;

        char cb;

    public:

        B():ib(0),cb('B') {}

        virtual void f() { cout << "B::f()" << endl;}

        virtual void Bf() { cout << "B::Bf()" << endl;}

};

class B1 :  public B

{

    public:

        int ib1;

        char cb1;

    public:

        B1():ib1(11),cb1('1') {}

        virtual void f() { cout << "B1::f()" << endl;}

        virtual void f1() { cout << "B1::f1()" << endl;}

        virtual void Bf1() { cout << "B1::Bf1()" << endl;}

};

class B2:  public B

{

    public:

        int ib2;

        char cb2;

    public:

        B2():ib2(12),cb2('2') {}

        virtual void f() { cout << "B2::f()" << endl;}

        virtual void f2() { cout << "B2::f2()" << endl;}

        virtual void Bf2() { cout << "B2::Bf2()" << endl;}

};

class D : public B1, public B2

{

    public:

        int id;

        char cd;

    public:

        D():id(100),cd('D') {}

        virtual void f() { cout << "D::f()" << endl;}

        virtual void f1() { cout << "D::f1()" << endl;}

        virtual void f2() { cout << "D::f2()" << endl;}

        virtual void Df() { cout << "D::Df()" << endl;}

};

我们用来存取子类内存布局的代码如下所示：（在VC++ 2003和G++ 3.4.4下）

    typedef void(\*Fun)(void);

    int\*\* pVtab = NULL;

    Fun pFun = NULL;

    D d;

    pVtab = (int\*\*)&d;

    cout << "[0] D::B1::\_vptr->" << endl;

    pFun = (Fun)pVtab[0][0];

    cout << "     [0] ";    pFun();

    pFun = (Fun)pVtab[0][1];

    cout << "     [1] ";    pFun();

    pFun = (Fun)pVtab[0][2];

    cout << "     [2] ";    pFun();

    pFun = (Fun)pVtab[0][3];

    cout << "     [3] ";    pFun();

    pFun = (Fun)pVtab[0][4];

    cout << "     [4] ";    pFun();

    pFun = (Fun)pVtab[0][5];

    cout << "     [5] 0x" << pFun << endl;

    cout << "[1] B::ib = " << (int)pVtab[1] << endl;

    cout << "[2] B::cb = " << (char)pVtab[2] << endl;

    cout << "[3] B1::ib1 = " << (int)pVtab[3] << endl;

    cout << "[4] B1::cb1 = " << (char)pVtab[4] << endl;

    cout << "[5] D::B2::\_vptr->" << endl;

    pFun = (Fun)pVtab[5][0];

    cout << "     [0] ";    pFun();

    pFun = (Fun)pVtab[5][1];

    cout << "     [1] ";    pFun();

    pFun = (Fun)pVtab[5][2];

    cout << "     [2] ";    pFun();

    pFun = (Fun)pVtab[5][3];

    cout << "     [3] ";    pFun();

    pFun = (Fun)pVtab[5][4];

    cout << "     [4] 0x" << pFun << endl;

    cout << "[6] B::ib = " << (int)pVtab[6] << endl;

    cout << "[7] B::cb = " << (char)pVtab[7] << endl;

    cout << "[8] B2::ib2 = " << (int)pVtab[8] << endl;

    cout << "[9] B2::cb2 = " << (char)pVtab[9] << endl;

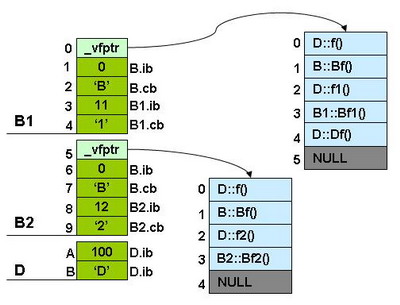
    cout << "[10] D::id = " << (int)pVtab[10] << endl;

    cout << "[11] D::cd = " << (char)pVtab[11] << endl;

程序运行结果如下：

|  |  |
| --- | --- |
| **GCC 3.4.4** | **VC++ 2003** |
| **[0] D::B1::\_vptr->**  **[0] D::f()**  **[1] B::Bf()**  **[2] D::f1()**  **[3] B1::Bf1()**  **[4] D::f2()**  **[5] 0x1**  **[1] B::ib = 0**  **[2] B::cb = B**  **[3] B1::ib1 = 11**  **[4] B1::cb1 = 1**  **[5] D::B2::\_vptr->**  **[0] D::f()**  **[1] B::Bf()**  **[2] D::f2()**  **[3] B2::Bf2()**  **[4] 0x0**  **[6] B::ib = 0**  **[7] B::cb = B**  **[8] B2::ib2 = 12**  **[9] B2::cb2 = 2**  **[10] D::id = 100**  **[11] D::cd = D** | **[0] D::B1::\_vptr->**  **[0] D::f()**  **[1] B::Bf()**  **[2] D::f1()**  **[3] B1::Bf1()**  **[4] D::Df()**  **[5] 0x00000000**  **[1] B::ib = 0**  **[2] B::cb = B**  **[3] B1::ib1 = 11**  **[4] B1::cb1 = 1**  **[5] D::B2::\_vptr->**  **[0] D::f()**  **[1] B::Bf()**  **[2] D::f2()**  **[3] B2::Bf2()**  **[4] 0x00000000**  **[6] B::ib = 0**  **[7] B::cb = B**  **[8] B2::ib2 = 12**  **[9] B2::cb2 = 2**  **[10] D::id = 100**  **[11] D::cd = D** |

下面是对于子类实例中的虚函数表的图：



我们可以看见，最顶端的父类B其成员变量存在于B1和B2中，并被D给继承下去了。而在D中，其有B1和B2的实例，于是B的成员在D的实例中存在两份，一份是B1继承而来的，另一份是B2继承而来的。所以，如果我们使用以下语句，则会产生二义性编译错误：

D d;

d.ib = 0;               //二义性错误

d.B1::ib = 1;           //正确

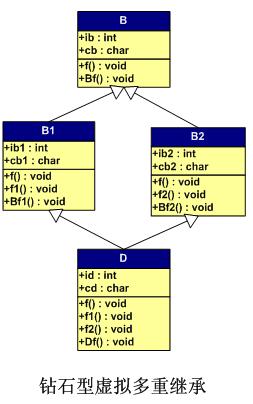
d.B2::ib = 2;           //正确

注意，上面例程中的最后两条语句存取的是两个变量。虽然我们消除了二义性的编译错误，但B类在D中还是有两个实例，这种继承造成了数据的重复，我们叫这种继承为重复继承。重复的基类数据成员可能并不是我们想要的。所以，C++引入了虚基类的概念。

## 钻石型多重虚拟继承

虚拟继承的出现就是为了解决重复继承中多个间接父类的问题的。钻石型的结构是其最经典的结构。也是我们在这里要讨论的结构：

上述的“重复继承”只需要把B1和B2继承B的语法中加上virtual 关键，就成了虚拟继承，其继承图如下所示：



上图和前面的“重复继承”中的类的内部数据和接口都是完全一样的，只是我们采用了虚拟继承：其省略后的源码如下所示：

class B {……};

class B1 : **virtual** public B{……};

class B2: **virtual** public B{……};

class D : public B1, public B2{ …… };

在查看D之前，我们先看一看**单一虚拟继承**的情况。下面是一段在VC++2003下的[**测试**](http://lib.csdn.net/base/softwaretest)程序：（因为VC++和GCC的内存而局上有一些细节上的不同，所以这里只给出VC++的程序，GCC下的程序大家可以根据我给出的程序自己仿照着写一个去试一试）：

    int\*\* pVtab = NULL;

    Fun pFun = NULL;

    B1 bb1;

    pVtab = (int\*\*)&bb1;

    cout << "[0] B1::\_vptr->" << endl;

    pFun = (Fun)pVtab[0][0];

    cout << "     [0] ";

    pFun(); //B1::f1();

    cout << "     [1] ";

    pFun = (Fun)pVtab[0][1];

    pFun(); //B1::bf1();

    cout << "     [2] ";

    cout << pVtab[0][2] << endl;

    cout << "[1] = 0x";

    cout << (int\*)\*((int\*)(&bb1)+1) <<endl; //B1::ib1

    cout << "[2] B1::ib1 = ";

    cout << (int)\*((int\*)(&bb1)+2) <<endl; //B1::ib1

    cout << "[3] B1::cb1 = ";

    cout << (char)\*((int\*)(&bb1)+3) << endl; //B1::cb1

    cout << "[4] = 0x";

    cout << (int\*)\*((int\*)(&bb1)+4) << endl; //NULL

    cout << "[5] B::\_vptr->" << endl;

    pFun = (Fun)pVtab[5][0];

    cout << "     [0] ";

    pFun(); //B1::f();

    pFun = (Fun)pVtab[5][1];

    cout << "     [1] ";

    pFun(); //B::Bf();

    cout << "     [2] ";

    cout << "0x" << (Fun)pVtab[5][2] << endl;

    cout << "[6] B::ib = ";

    cout << (int)\*((int\*)(&bb1)+6) <<endl; //B::ib

    cout << "[7] B::cb = ";

其运行结果如下（我结出了GCC的和VC++2003的对比）：

|  |  |
| --- | --- |
| **GCC 3.4.4** | **VC++ 2003** |
| **[0] B1::\_vptr ->**  **[0] : B1::f()**  **[1] : B1::f1()**  **[2] : B1::Bf1()**  **[3] : 0**  **[1] B1::ib1 : 11**  **[2] B1::cb1 : 1**  **[3] B::\_vptr ->**  **[0] : B1::f()**  **[1] : B::Bf()**  **[2] : 0**  **[4] B::ib : 0**  **[5] B::cb : B**  **[6] NULL : 0** | **[0] B1::\_vptr->**  **[0] B1::f1()**  **[1] B1::Bf1()**  **[2] 0**  **[1] = 0x00454310 该地址取值后是-4**  **[2] B1::ib1 = 11**  **[3] B1::cb1 = 1**  **[4] = 0x00000000**  **[5] B::\_vptr->**  **[0] B1::f()**  **[1] B::Bf()**  **[2] 0x00000000**  **[6] B::ib = 0**  **[7] B::cb = B** |

这里，大家可以自己对比一下。关于细节上，我会在后面一并再说。

下面的测试程序是看子类D的内存布局，同样是VC++ 2003的（因为VC++和GCC的内存布局上有一些细节上的不同，而VC++的相对要清楚很多，所以这里只给出VC++的程序，GCC下的程序大家可以根据我给出的程序自己仿照着写一个去试一试）：

    D d;

    pVtab = (int\*\*)&d;

    cout << "[0] D::B1::\_vptr->" << endl;

    pFun = (Fun)pVtab[0][0];

    cout << "     [0] ";    pFun(); //D::f1();

    pFun = (Fun)pVtab[0][1];

    cout << "     [1] ";    pFun(); //B1::Bf1();

    pFun = (Fun)pVtab[0][2];

    cout << "     [2] ";    pFun(); //D::Df();

    pFun = (Fun)pVtab[0][3];

    cout << "     [3] ";

    cout << pFun << endl;

    //cout << pVtab[4][2] << endl;

    cout << "[1] = 0x";

    cout <<  (int\*)((&dd)+1) <<endl; //????

    cout << "[2] B1::ib1 = ";

    cout << \*((int\*)(&dd)+2) <<endl; //B1::ib1

    cout << "[3] B1::cb1 = ";

    cout << (char)\*((int\*)(&dd)+3) << endl; //B1::cb1

    //---------------------

    cout << "[4] D::B2::\_vptr->" << endl;

    pFun = (Fun)pVtab[4][0];

    cout << "     [0] ";    pFun(); //D::f2();

    pFun = (Fun)pVtab[4][1];

    cout << "     [1] ";    pFun(); //B2::Bf2();

    pFun = (Fun)pVtab[4][2];

    cout << "     [2] ";

    cout << pFun << endl;

    cout << "[5] = 0x";

    cout << \*((int\*)(&dd)+5) << endl; // ???

    cout << "[6] B2::ib2 = ";

    cout << (int)\*((int\*)(&dd)+6) <<endl; //B2::ib2

    cout << "[7] B2::cb2 = ";

    cout << (char)\*((int\*)(&dd)+7) << endl; //B2::cb2

    cout << "[8] D::id = ";

    cout << \*((int\*)(&dd)+8) << endl; //D::id

    cout << "[9] D::cd = ";

    cout << (char)\*((int\*)(&dd)+9) << endl;//D::cd

    cout << "[10]  = 0x";

    cout << (int\*)\*((int\*)(&dd)+10) << endl;

    //---------------------

    cout << "[11] D::B::\_vptr->" << endl;

    pFun = (Fun)pVtab[11][0];

    cout << "     [0] ";    pFun(); //D::f();

    pFun = (Fun)pVtab[11][1];

    cout << "     [1] ";    pFun(); //B::Bf();

    pFun = (Fun)pVtab[11][2];

    cout << "     [2] ";

    cout << pFun << endl;

    cout << "[12] B::ib = ";

    cout << \*((int\*)(&dd)+12) << endl; //B::ib

    cout << "[13] B::cb = ";

    cout << (char)\*((int\*)(&dd)+13) <<endl;//B::cb

下面给出运行后的结果（分VC++和GCC两部份）

|  |  |
| --- | --- |
| **GCC 3.4.4** | **VC++ 2003** |
| **[0] B1::\_vptr ->**  **[0] : D::f()**  **[1] : D::f1()**  **[2] : B1::Bf1()**  **[3] : D::f2()**  **[4] : D::Df()**  **[5] : 1**  **[1] B1::ib1 : 11**  **[2] B1::cb1 : 1**  **[3] B2::\_vptr ->**  **[0] : D::f()**  **[1] : D::f2()**  **[2] : B2::Bf2()**  **[3] : 0**  **[4] B2::ib2 : 12**  **[5] B2::cb2 : 2**  **[6] D::id : 100**  **[7] D::cd : D**  **[8] B::\_vptr ->**  **[0] : D::f()**  **[1] : B::Bf()**  **[2] : 0**  **[9] B::ib : 0**  **[10] B::cb : B**  **[11] NULL : 0** | **[0] D::B1::\_vptr->**  **[0] D::f1()**  **[1] B1::Bf1()**  **[2] D::Df()**  **[3] 00000000**  **[1] = 0x0013FDC4   该地址取值后是-4**  **[2] B1::ib1 = 11**  **[3] B1::cb1 = 1**  **[4] D::B2::\_vptr->**  **[0] D::f2()**  **[1] B2::Bf2()**  **[2] 00000000**  **[5] = 0x4539260    该地址取值后是-4**  **[6] B2::ib2 = 12**  **[7] B2::cb2 = 2**  **[8] D::id = 100**  **[9] D::cd = D**  **[10]  = 0x00000000**  **[11] D::B::\_vptr->**  **[0] D::f()**  **[1] B::Bf()**  **[2] 00000000**  **[12] B::ib = 0**  **[13] B::cb = B** |

关于虚拟继承的运行结果我就不画图了（前面的作图已经让我产生了很严重的厌倦感，所以就偷个懒了，大家见谅了）

在上面的输出结果中，我用不同的颜色做了一些标明。我们可以看到如下的几点：

1）无论是GCC还是VC++，除了一些细节上的不同，其大体上的对象布局是一样的。也就是说，先是B1（黄色），然后是B2（绿色），接着是D（灰色），而B这个超类（青蓝色）的实例都放在最后的位置。

2）关于虚函数表，尤其是第一个虚表，GCC和VC++有很重大的不一样。但仔细看下来，还是VC++的虚表比较清晰和有逻辑性。

3）VC++和GCC都把B这个超类放到了最后，而VC++有一个NULL分隔符把B和B1和B2的布局分开。GCC则没有。

4）VC++中的内存布局有两个地址我有些不是很明白，在其中我用红色标出了。取其内容是-4。接道理来说，这个指针应该是指向B类实例的内存地址（这个做法就是为了保证重复的父类只有一个实例的技术）。但取值后却不是。这点我目前还并不太清楚，还向大家请教。

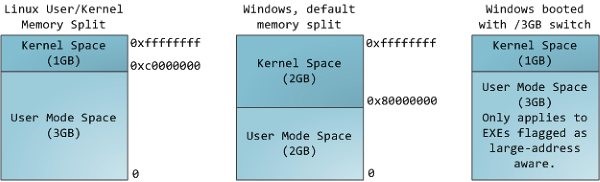
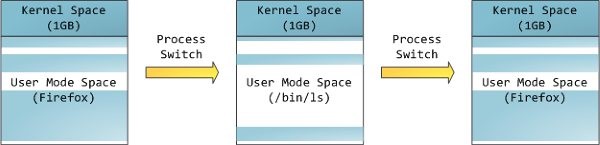
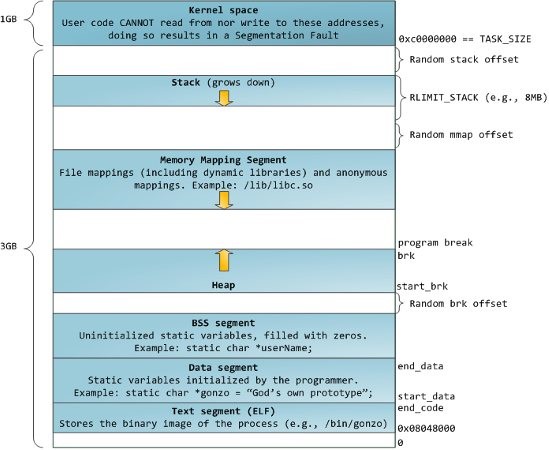
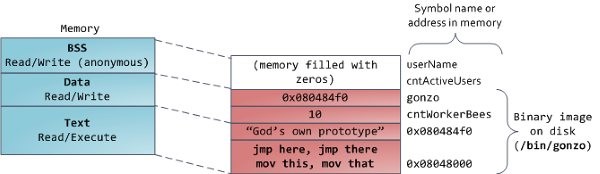
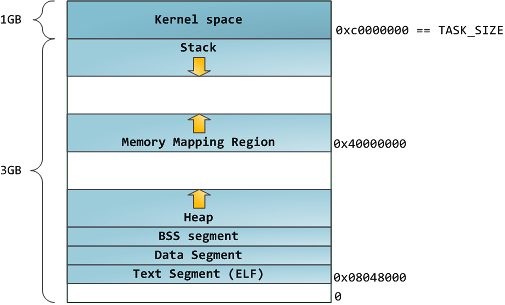
5）GCC的内存布局中在B1和B2中则没有指向B的指针。这点可以理解，编译器可以通过计算B1和B2的size而得出B的偏移量。

## 结束语

C++这门语言是一门比较复杂的语言，对于程序员来说，我们似乎永远摸不清楚这门语言背着我们在干了什么。需要熟悉这门语言，我们就必需要了解C++里面的那些东西，需要我们去了解他后面的内存对象。这样我们才能真正的了解C++，从而能够更好的使用C++这门最难的编程语言。

在文章束之前还是介绍一下自己吧。我从事软件研发有十个年头了，目前是软件开发技术主管，技术方面，主攻Unix/C/C++，比较喜欢网络上的技术，比如分布式计算，网格计算，P2P，Ajax等一切和互联网相关的东西。管理方面比较擅长于团队建设，技术趋势分析，项目管理。欢迎大家和我交流，我的MSN和Email是：[haoel@hotmail.com](mailto:haoel@hotmail.com)

# [linux系统进程的内存布局](http://www.cnblogs.com/diegodu/p/4552490.html)

内存管理模块是操作系统的心脏；它对应用程序和系统管理非常重要。今后的几篇文章中，我将着眼于实际的内存问题，但也不避讳其中的技术内幕。由于不少概念是通用的，所以文中大部分例子取自32位x86平台的Linux和Windows系统。本系列第一篇文章讲述应用程序的内存布局。  
  
    在多任务操作系统中的每一个进程都运行在一个属于它自己的内存沙盘中。这个沙盘就是虚拟地址空间（virtual address space），在32位模式下它总是一个4GB的内存地址块。这些虚拟地址通过页表（page table）映射到物理内存，页表由操作系统维护并被处理器引用。每一个进程拥有一套属于它自己的页表，但是还有一个隐情。只要虚拟地址被使能，那么它就会作用于这台机器上运行的所有软件，包括内核本身。因此一部分虚拟地址必须保留给内核使用：  
  
  
这并不意味着内核使用了那么多的物理内存，仅表示它可支配这么大的地址空间，可根据内核需要，将其映射到物理内存。内核空间在页表中拥有较高的特权级（ring 2或以下），因此只要用户态的程序试图访问这些页，就会导致一个页错误（page fault）。在Linux中，内核空间是持续存在的，并且在所有进程中都映射到同样的物理内存。内核代码和数据总是可寻址的，随时准备处理中断和系统调用。与此相反，用户模式地址空间的映射随进程切换的发生而不断变化：  
  
  
色区域表示映射到物理内存的虚拟地址，而白色区域表示未映射的部分。在上面的例子中，Firefox使用了相当多的虚拟地址空间，因为它是传说中的吃内存大户。地址空间中的各个条带对应于不同的内存段（memory segment），如：堆、栈之类的。记住，这些段只是简单的内存地址范围，与Intel处理器的段没有关系。不管怎样，下面是一个Linux进程的标准的内存段布局：  
  
  
当计算机开心、安全、可爱、正常的运转时，几乎每一个进程的各个段的起始虚拟地址都与上图完全一致，这也给远程发掘程序安全漏洞打开了方便之门。一个发掘过程往往需要引用绝对内存地址：栈地址，库函数地址等。远程攻击者必须依赖地址空间布局的一致性，摸索着选择这些地址。如果让他们猜个正着，有人就会被整了。因此，地址空间的随机排布方式逐渐流行起来。Linux 通过对[栈](http://lxr.linux.no/linux+v2.6.28.1/fs/binfmt_elf.c#L542)[内存映射段](http://lxr.linux.no/linux+v2.6.28.1/arch/x86/mm/mmap.c#L84)、[堆](http://lxr.linux.no/linux+v2.6.28.1/arch/x86/kernel/process_32.c#L729)的起始地址加上随机的偏移量来打乱布局。不幸的是，32 位地址空间相当紧凑，给随机化所留下的空当不大，[削弱了这种技巧的效果](http://www.stanford.edu/~blp/papers/asrandom.pdf)。  
  
进程地址空间中最顶部的段是栈，大多数编程语言将之用于存储局部变量和函数参数。调用一个方法或函数会将一个新的栈桢（stack frame）压入栈中。栈桢在函数返回时被清理。也许是因为数据严格的遵从[LIFO](http://en.wikipedia.org/wiki/Lifo)的顺序，这个简单的设计意味着不必使用复杂的数据结构来追踪栈的内容，只需要一个简单的指针指向栈的顶端即可。因此压栈（pushing）和退栈（popping）过程非常迅速、准确。另外，持续的重用栈空间有助于使活跃的栈内存保持在[CPU缓存中](http://blog.csdn.net/drshenlei/archive/2009/06/17/4277959.aspx)，从而加速访问。进程中的每一个线程都有属于自己的栈。  
  
通过不断向栈中压入的数据，超出其容量就有会耗尽栈所对应的内存区域。这将触发一个页故障（page fault），并被 Linux 的[expand\_stack()](http://lxr.linux.no/linux+v2.6.28/mm/mmap.c#L1716)处理，它会调用[acct\_stack\_growth()](http://lxr.linux.no/linux+v2.6.28/mm/mmap.c#L1544)来检查是否还有合适的地方用于栈的增长。如果栈的大小低于RLIMIT\_STACK（通常是8MB），那么一般情况下栈会被加长，程序继续愉快的运行，感觉不到发生了什么事情。这是一种将栈扩展至所需大小的常规机制。然而，如果达到了最大的栈空间大小，就会栈溢出（stack overflow），程序收到一个段错误（Segmentation Fault）。当映射了的栈区域扩展到所需的大小后，它就不会再收缩回去，即使栈不那么满了。这就好比联邦预算，它总是在增长的。  
  
动态栈增长是唯一一种访问未映射内存区域（图中白色区域）而被允许的情形。其它任何对未映射内存区域的访问都会触发页故障，从而导致段错误。一些被映射的区域是只读的，因此企图写这些区域也会导致段错误。  
  
在栈的下方，是我们的内存映射段。此处，内核将文件的内容直接映射到内存。任何应用程序都可以通过 Linux 的 [mmap()](http://www.kernel.org/doc/man-pages/online/pages/man2/mmap.2.html)系统调用（[实现](http://lxr.linux.no/linux+v2.6.28.1/arch/x86/kernel/sys_i386_32.c#L27)）或 Windows 的 [CreateFileMapping()](http://msdn.microsoft.com/en-us/library/aa366537%28VS.85%29.aspx)/[MapViewOfFile()](http://msdn.microsoft.com/en-us/library/aa366761%28VS.85%29.aspx)请求这种映射。内存映射是一种方便高效的文件 I/O 方式，所以它被用于加载动态库。创建一个不对应于任何文件的匿名内存映射也是可能的，此方法用于存放程序的数据。在 Linux 中，如果你通过 [malloc()](http://www.kernel.org/doc/man-pages/online/pages/man3/malloc.3.html)请求一大块内存，C 运行库将会创建这样一个匿名映射而不是使用堆内存。‘大块’意味着比MMAP\_THRESHOLD 还大，缺省是 128KB ，可以通过[mallopt()](http://www.kernel.org/doc/man-pages/online/pages/man3/undocumented.3.html)调整。  
  
说到堆，它是接下来的一块地址空间。与栈一样，堆用于运行时内存分配；但不同点是，堆用于存储那些生存期与函数调用无关的数据。大部分语言都提供了堆管理功能。因此，满足内存请求就成了语言运行时库及内核共同的任务。在 C 语言中，堆分配的接口是[malloc()](http://www.kernel.org/doc/man-pages/online/pages/man3/malloc.3.html)系列函数，而在具有垃圾收集功能的语言（如 C# ）中，此接口是 new 关键字。  
  
如果堆中有足够的空间来满足内存请求，它就可以被语言运行时库处理而不需要内核参与。否则，堆会被扩大，通过[brk()](http://www.kernel.org/doc/man-pages/online/pages/man2/brk.2.html)系统调用（[实现](http://lxr.linux.no/linux+v2.6.28.1/mm/mmap.c#L248)）来分配请求所需的内存块。堆管理是很[复杂](http://g.oswego.edu/dl/html/malloc.html)的，需要精细的算法，应付我们程序中杂乱的分配模式，优化速度和内存使用效率。处理一个堆请求所需的时间会大幅度的变动。实时系统通过[特殊目的分配器](http://rtportal.upv.es/rtmalloc/)来解决这个问题。堆也可能会变得零零碎碎，如下图所示：  
  
  
最后，我们来看看最底部的内存段：BSS，数据段，代码段。在C语言中，BSS和数据段保存的都是静态（全局）变量的内容。区别在于BSS保存的是未被初始化的静态变量内容，它们的值不是直接在程序的源代码中设定的。BSS内存区域是匿名的：它不映射到任何文件。如果你写static int cntActiveUsers，则cntActiveUsers的内容就会保存在BSS中。  
  
    另一方面，数据段保存在源代码中已经初始化了的静态变量内容。这个内存区域不是匿名的。它映射了一部分的程序二进制镜像，也就是源代码中指定了初始值的静态变量。所以，如果你写static int cntWorkerBees = 10，则cntWorkerBees的内容就保存在数据段中了，而且初始值为10。尽管数据段映射了一个文件，但它是一个私有内存映射，这意味着更改此处的内存不会影响到被映射的文件。也必须如此，否则给全局变量赋值将会改动你硬盘上的二进制镜像，这是不可想象的。  
  
    下图中数据段的例子更加复杂，因为它用了一个指针。在此情况下，指针gonzo（4字节内存地址）本身的值保存在数据段中。而它所指向的实际字符串则不在这里。这个字符串保存在代码段中，代码段是只读的，保存了你全部的代码外加零零碎碎的东西，比如字符串字面值。代码段将你的二进制文件也映射到了内存中，但对此区域的写操作都会使你的程序收到段错误。这有助于防范指针错误，虽然不像在C语言编程时就注意防范来得那么有效。下图展示了这些段以及我们例子中的变量：  
  
你可以通过阅读文件/proc/pid\_of\_process/maps来检验一个Linux进程中的内存区域。记住一个段可能包含许多区域。比如，每个内存映射文件在mmap段中都有属于自己的区域，动态库拥有类似BSS和数据段的额外区域。下一篇文章讲说明这些“区域”（area）的真正含义。有时人们提到“数据段”，指的就是全部的数据段 + BSS + 堆。  
  
你可以通过[nm](http://manpages.ubuntu.com/manpages/intrepid/en/man1/nm.1.html)和[objdump](http://manpages.ubuntu.com/manpages/intrepid/en/man1/objdump.1.html)命令来察看二进制镜像，打印其中的符号，它们的地址，段等信息。最后需要指出的是，前文描述的虚拟地址布局在Linux 中是一种“灵活布局”（flexible layout），而且以此作为默认方式已经有些年头了。它假设我们有值 RLIMIT\_STACK。当情况不是这样时， Linux 退回使用“经典布局”（classic layout），如下图所示：  
  
对虚拟地址空间的布局就讲这些吧。下一篇文章将讨论内核是如何跟踪这些内存区域的。我们会分析内存映射，看看文件的读写操作是如何与之关联的，以及内存使用概况的含义。

对任何一个普通C++程序来讲，它都会涉及到5种不同的数据段。常用的几个数据段种包含有“程序代码段”、“程序数据段”、“程序堆栈段”等。不错，这几种数据段都在其中，但除了以上几种数据段之外，进程还另外包含两种数据段。下面我们来简单归纳一下进程对应的内存空间中所包含的5种不同的数据区。

代码段：代码段是用来存放可执行文件的操作指令，也就是说是它是可执行程序在内存种的镜像。代码段需要防止在运行时被非法修改，所以只准许读取操作，而不允许写入（修改）操作——它是不可写的。

数据段：数据段用来存放可执行文件中已初始化全局变量，换句话说就是存放程序静态分配的变量和全局变量。

BSS段：BSS段包含了程序中未初始化全局变量，在内存中bss段全部置零。

堆（heap）：堆是用于存放进程运行中被动态分配的内存段，它大小并不固定，可动态扩张或缩减。当进程调用malloc/new等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用free等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）

栈：栈是用户存放程序临时创建的局部变量，也就是说我们函数括弧“{}”中定义的变量（但不包括static声明的变量，static意味这在数据段中存放变量）。除此以外在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也回被存放回栈中。由于栈的先进先出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上将我们可以把堆栈看成一个临时数据寄存、交换的内存区。

我们要知道，栈中存放的是一个个被调函数所对应的堆栈帧，当函数fun1被调用，则fun1的堆栈帧入栈，fun1返回时，fun1的堆栈帧出栈。什么是堆栈帧呢，堆栈帧其实就是保存被调函数返回时下一条执行指令的指针、主调函数的堆栈帧的指针、主调函数传递给被调函数的实参(如果有的话)、被调函数的局部变量等信息的一个结构。

首先，我们要说明的是如何区分每个堆栈帧，或者说，如何知道我现在在使用哪个堆栈帧。和栈密切相关的有2个寄存器，一个是ebp,一个是esp,前者可以叫作栈基址指针，后者可以叫栈顶指针。对于一个堆栈帧来说，ebp也叫堆栈帧指针,它永远指向这个堆栈帧的某个固定位置(见上图)，所以可以根据ebp来表示一个堆栈帧，可以通过对ebp的偏移加减，来在堆栈帧中来来回回的访问。esp则是随着push和pop而不断移动。因此根据esp来对堆栈帧进行操作。

再来讲一下上图，一个堆栈帧的最顶部，是实参，然后是return address,这个值是由主调函数中的call命令在call调用时自动压入的，不需要我们关心，previousframe pointer,就是主调函数的堆栈帧指针，也就是主调函数的ebp值。ebp偏移为正的都是被调函数的局部变量。

# linux查看修改线程默认栈空间大小（ulimit -s）

linux查看修改线程默认栈空间大小 ulimit -s

1、通过命令 ulimit -s 查看linux的默认栈空间大小，默认情况下 为10240 即10M

2、通过命令 ulimit -s 设置大小值 临时改变栈空间大小：ulimit -s 102400， 即修改为100M

3、可以在/etc/rc.local 内 加入 ulimit -s 102400 则可以开机就设置栈空间大小

4、在/etc/security/limits.conf 中也可以改变栈空间大小：

#<domain> <type> <item> <value>

\* soft stack 102400

重新登录，执行ulimit -s 即可看到改为102400 即100M

# Linux的nm查看动态和静态库中的符号

## 功能

列出.o .a .so中的符号信息，包括诸如符号的值，符号类型及符号名称等。所谓符号，通常指定义出的函数，全局变量等等。

## 使用

**nm [option(s)] [file(s)]**

有用的**options**:

* -A 在每个符号信息的前面打印所在对象文件名称；
* -C 输出demangle过了的符号名称；
* -D 打印动态符号；
* -l 使用对象文件中的调试信息打印出所在源文件及行号；
* -n 按照地址/符号值来排序；
* -u 打印出那些未定义的符号；

常见的**符号类型**:

* A 该符号的值在今后的链接中将不再改变；
* B 该符号放在BSS段中，通常是那些未初始化的全局变量；
* D 该符号放在普通的数据段中，通常是那些已经初始化的全局变量；
* T 该符号放在代码段中，通常是那些全局非静态函数；
* U 该符号未定义过，需要自其他对象文件中链接进来；
* W 未明确指定的弱链接符号；同链接的其他对象文件中有它的定义就用上，否则就用一个系统特别指定的默认值。

注意几点：

* -C 总是适用于c++编译出来的对象文件。还记得c++中有重载么？为了区分重载函数，c++编译器会将函数返回值/参数等信息附加到函数名称中去形成一个mangle过的符号，那用这个选项列出符号的时候，做一个逆操作，输出那些原始的、我们可理解的符号名称。
* 使用 -l 时，必须保证你的对象文件中带有符号调式信息，这一般要求你在编译的时候指定一个 -g 选项，见 [Linux:Gcc](http://www.juliantec.info/julwiki/index.php?title=Linux:Gcc&action=edit&redlink=1)。
* 使用nm前，最好先用[Linux:File](http://www.juliantec.info/wiki/Linux:File)查看对象文件所属处理器架构，然后再用相应交叉版本的nm工具。

## 举例

更详细的内容见man page。这里举例说明：

**nm -u hello.o**

显示hello.o 中的未定义符号，需要和其他对象文件进行链接.

**nm -A /usr/lib/\* 2>/dev/null | grep "T memset"**

在 /usr/lib/ 目录下找出哪个库文件定义了memset函数.

# [malloc()\_分配内存，会用到brk或mmap2系统调用](http://blog.chinaunix.net/uid-26969690-id-3719687.html)

**malloc()分配内存的时候，可能调用brk()，也可能调用mmap2()。**

**分配：1、**即分配一块小型内存(小于或等于128kb)，malloc()会调用brk()调高断点(brk是将数据段(.data)的最高地址指针\_edata往高地址推)，**分配的内存在堆区域**，**2、**当分配一块大型内存(大于128kb),malloc()会调用mmap2()分配一块内存（mmap是在进程的虚拟地址空间中（一般是堆和栈中间）找一块空闲的）**（一般是堆和栈中间区域）**。

**malloc分配的内存只分配了虚拟地址空间，当在第一次访问的时候，发生缺页中断，操作系统负责分配物理内存，并建立虚拟内存和物理内存之间的映射关系。**

**释放：**同样的，

**1、**munmap释放内存映射方式分配的内存之后，内存**马上**会被系统收回（不会产生内存碎片）， **2、**释放brk分配的一块内存，并不会马上被系统回收(因为移动\_edata指针时，只有最高位对应的虚拟内存释放了，才能往前移动，当然，这块内存可以被再分配了。参考<http://blog.163.com/xychenbaihu@yeah/blog/getBlog.do?fromString=blogmodule>)，glibc会保留它以供下一次malloc()使用，glibc有一套自己的内存管理机制。

brk和sbrk主要的工作是实现虚拟内存到内存的映射

# [Linux中brk()系统调用,sbrk(),mmap(),malloc(),calloc()的异同](http://blog.csdn.net/kobbee9/article/details/7397010)

brk和sbrk主要的工作是实现虚拟内存到内存的映射.在GNUC中,内存分配是这样的:  
       每个进程可访问的虚拟内存空间为3G，但在程序编译时，不可能也没必要为程序分配这么大的空间，只分配并不大的数据段空间，程序中动态分配的空间就是从这一块分配的。如果这块空间不够，malloc函数族（realloc，calloc等）就调用sbrk函数将数据段的下界移动，sbrk函数在内核的管理下将虚拟地址空间映射到内存，供malloc函数使用。（参见[**Linux**](http://lib.csdn.net/base/linux)内核情景分析）

#include <unistd.h>

       int brk(void \*end\_data\_segment);

       void \*sbrk(ptrdiff\_t increment);

DESCRIPTION  
       brk   sets   the   end   of   the   data   segment   to   the value specified by end\_data\_segment, when that value is reasonable, the system   does   have enough   memory   and   the process does not exceed its max data size (see setrlimit(2)).

       sbrk increments the program's data   space   by   increment   bytes.    sbrk isn't a system call, it is just a C library wrapper.   Calling sbrk with an increment of 0 can be used to find the current location of the   program break.

RETURN VALUE  
       On   success,   brk returns zero, and sbrk returns a pointer to the start of the new area.   On error, -1 is returned, and errno is set to ENOMEM.

sbrk不是系统调用，是C库函数。系统调用通常提供一种最小功能，而库函数通常提供比较复杂的功能。

在Linux系统上，程序被载入内存时，内核为用户进程地址空间建立了代码段、数据段和堆栈段，在数据段与堆栈段之间的空闲区域用于动态内存分配。

内核[**数据结构**](http://lib.csdn.net/base/datastructure)mm\_struct中的成员变量start\_code和end\_code是进程代码段的起始和终止地址，start\_data和 end\_data是进程数据段的起始和终止地址，start\_stack是进程堆栈段起始地址，start\_brk是进程动态内存分配起始地址（堆的起始地址），还有一个 brk（堆的当前最后地址），就是动态内存分配当前的终止地址。

[**C语言**](http://lib.csdn.net/base/c)的动态内存分配基本函数是malloc()，在Linux上的基本实现是通过内核的brk系统调用。brk()是一个非常简单的系统调用，只是简单地改变mm\_struct结构的成员变量brk的值。

mmap系统调用实现了更有用的动态内存分配功能，可以将一个磁盘文件的全部或部分内容映射到用户空间中，进程读写文件的操作变成了读写内存的操作。在 linux/mm/mmap.c文件的do\_mmap\_pgoff()函数，是mmap系统调用实现的核心。do\_mmap\_pgoff()的代码，只是新建了一个vm\_area\_struct结构，并把file结构的参数赋值给其成员变量m\_file，并没有把文件内容实际装入内存。  
Linux内存管理的基本思想之一，是只有在真正访问一个地址的时候才建立这个地址的物理映射。

============================================================================  
C语言跟内存分配方式  
（1） 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。  
（2） 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。  
（3）从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

C语言跟内存申请相关的函数主要有 alloc,calloc,malloc,free,realloc,sbrk等.其中alloc是向栈申请内存,因此无需释放. malloc分配的内存是位于堆中的,并且没有初始化内存的内容,因此基本上malloc之后,调用函数memset来初始化这部分的内存空间.calloc则将初始化这部分的内存,设置为0. 而realloc则对malloc申请的内存进行大小的调整.申请的内存最终需要通过函数free来释放. 而sbrk则是增加数据段的大小;

malloc/calloc/free基本上都是C函数库实现的,跟OS无关.C函数库内部通过一定的结构来保存当前有多少可用内存.如果程序 malloc的大小超出了库里所留存的空间,那么将首先调用brk系统调用来增加可用空间,然后再分配空间.free时,释放的内存并不立即返回给os, 而是保留在内部结构中. 可以打个比方: brk类似于批发,一次性的向OS申请大的内存,而malloc等函数则类似于零售,满足程序运行时的要求.这套机制类似于缓冲.

使用这套机制的原因: 系统调用不能支持任意大小的内存分配(有的系统调用只支持固定大小以及其倍数的内存申请,这样的话,对于小内存的分配会造成浪费; 系统调用申请内存代价昂贵,涉及到用户态和核心态的转换.  
函数malloc()和calloc()都可以用来分配动态内存空间，但两者稍有区别。     
     malloc()函数有一个参数，即要分配的内存空间的大小：      
     void \*malloc(size\_t size);   
     calloc()函数有两个参数，分别为元素的数目和每个元素的大小，这两个参数的乘积就是要分配的内存空间的大小：     
     void \*calloc(size\_t numElements，size\_t sizeOfElement)；  
     如果调用成功，函数malloc()和calloc()都将返回所分配的内存空间的首地址。  
     malloc() 函数和calloc()函数的主要区别是前者不能初始化所分配的内存空间，而后者能。如果由malloc()函数分配的内存空间原来没有被使用过，则其中的每一位可能都是0；反之，如果这部分内存空间曾经被分配、释放和重新分配，则其中可能遗留各种各样的数据。也就是说，使用malloc()函数的程序开始时(内存空间还没有被重新分配)能正常运行，但经过一段时间后(内存空间已被重新分配)可能会出现问题。  
     calloc() 函数会将所分配的内存空间中的每一位都初始化为零，也就是说，如果你是为字符类型或整数类型的元素分配内存，那么这些元素将保证会被初始化为零；如果你是为指针类型的元素分配内存，那么这些元素通常(但无法保证)会被初始化为空指针；如果你是为实数类型的元素分配内存，那么这些元素可能(只在某些计算机中)会被初始化为浮点型的零。  
     malloc() 函数和calloc()函数的另一点区别是calloc()函数会返回一个由某种对象组成的数组，但malloc()函数只返回一个对象。为了明确是为一个数组分配内存空间，有些程序员会选用calloc()函数。但是，除了是否初始化所分配的内存空间这一点之外，绝大多数程序员认为以下两种函数调用方式没有区别：  
     calloc(numElements ，sizeOfElement)；  
     malloc(numElements \*sizeOfElement) ；  
     需要解释的一点是，理论上(按照ANSIC标准)指针的算术运算只能在一个指定的数组中进行，但是在实践中，即使C编译程序或翻译器遵循这种规定，许多C 程序还是冲破了这种限制。因此，尽管malloc()函数并不能返回一个数组，它所分配的内存空间仍然能供一个数组使用(对realloc()函数来说同样如此，尽管它也不能返回一个数组)。  
     总之，当你在calloc()函数和malloc()函数之间作选择时，你只需考虑是否要初始化所分配的内存空间，而不用考虑函数是否能返回一个数组。  
     当程序运行过程中malloc了,但是没有free的话,会造成内存泄漏.一部分的内存没有被使用,但是由于没有free,因此系统认为这部分内存还在使用,造成不断的向系统申请内存,是的系统可用内存不断减少.但是,内存泄漏仅仅指程序在运行时,程序退出时,OS将回收所有的资源.因此,适当的重起一下程序,有时候还是有点作用.

sbrk(int incr) 本函数用来增加分配给调用程序的数据段的空间数量,增加incr个字节的空间brk函数的原形是：int    brk(void    \*endds)   
   它的功能是：更改数据段空间的分配   
   char    \*p;   
   p=malloc(1);   
   这时p指向的内存空间大小是1    byte   
   brk(p+100)   
   这时p指向的内存空间大小是101    bytes

      程式分配虚拟内存也不是你要一个字节就给你一个字节，而是你要一个字节给你一个页面，因为映射物理内存时只能以页为单位。你要另一个字节时，它在这个页面的剩余空间给你。

注意大部份UNIX虚拟内存的使用是只增不减的。

CODE:malloc(32 \* 1024) --->;sbrk += 32 \* 1024  
free()    --->;sbrk 不减少。  
但如如果再来一次  
malloc(32 \* 1024) ---->;sbrk 也不增，使用原有空间.  
但对于LINUX来说它是要以内存的最大数收缩的；

CODE:<code>  
a = malloc(32 \* 1024) -->;sbrk += 32 \* 1024  
b = malloc(32 \* 1024) -->;sbrk += 32 \* 1024  
if(\*\*\*\*){  
free(b); --->;sbrk -= 32 \* 1024;  
}  
else{  
free(a); --->;sbrk 不减少。只是多了个空洞.  
}  
</code>  
CODE:<code>  
/\* linux kernel code \*/  
brk()  
/\*  
\*   sys\_brk() for the most part doesn't need the global kernel  
\*   lock, except when an application is doing something nasty  
\*   like trying to un-brk an area that has already been mapped  
\*   to a regular file.   in this case, the unmapping will need  
\*   to invoke file system routines that need the global lock.  
\*/  
asmlinkage unsigned long sys\_brk(unsigned long brk)  
{  
unsigned long rlim, retval;  
unsigned long newbrk, oldbrk;  
struct mm\_struct \*mm = current->;mm;  
down\_write(&mm->;mmap\_sem);  
if (brk < mm->;end\_code)  
goto out;  
newbrk = PAGE\_ALIGN(brk);  
oldbrk = PAGE\_ALIGN(mm->;brk);  
if (oldbrk == newbrk)  
goto set\_brk;  
    /\*\*\*\*\*\*虚拟内存在这里收缩\*\*\*\*\*\*/  
/\* Always allow shrinking brk. \*/  
if (brk <= mm->;brk) {  
if (!do\_munmap(mm, newbrk, oldbrk-newbrk))  
goto set\_brk;  
goto out;  
}  
/\* Check against rlimit.. \*/  
rlim = current->;rlim[RLIMIT\_DATA].rlim\_cur;  
if (rlim < RLIM\_INFINITY && brk - mm->;start\_data >; rlim)  
goto out;  
/\* Check against existing mmap mappings. \*/  
if (find\_vma\_intersection(mm, oldbrk, newbrk+PAGE\_SIZE))  
goto out;  
/\* Check if we have enough memory.. \*/  
if (!vm\_enough\_memory((newbrk-oldbrk) >;>; PAGE\_SHIFT))  
goto out;  
/\* Ok, looks good - let it rip. \*/  
if (do\_brk(oldbrk, newbrk-oldbrk) != oldbrk)  
goto out;  
set\_brk:  
mm->;brk = brk;  
out:  
retval = mm->;brk; /\*\*\*\*这就是返回值\*\*\*\*\*/  
up\_write(&mm->;mmap\_sem);  
return retval;  
}  
</code>  
在LINUX中sbrk(0)能返回比较精确的虚拟内存使用情况，  
在SOLARIS/HP中sbrk(0)返回以页为单位的虚拟内存使用情况。使用sbrk(0)来返回程式当前使用了多少内存。

<code>  
main(){  
int start,end;  
start = sbrk(0);  
....  
malloc(\*\*\*);  
....  
end = sbrk(0);  
printf("hello I used %d vmemory",end - start);  
}

</code

# 虚函数表放在什么地方？

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。

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

# 智能指针的原理

智能指针是通过引用计数器实现的。

两个智能指针交叉引用会出现什么问题？会出现引用失败，造成内存泄漏。

## c++智能指针介绍

由于 C++ 语言没有自动内存回收机制，程序员每次 new 出来的内存都要手动 delete，比如流程太复杂，最终导致没有 delete，异常导致程序过早退出，没有执行 delete 的情况并不罕见，并造成内存泄露。如此c++引入智能指针 ，智能指针即是C++ RAII的一种应用，可用于动态资源管理，资源即对象的管理策略。 智能指针在 <memory> 标头文件的 **std** 命名空间中定义。 它们对 RAII 或获取资源即初始化编程惯用法至关重要。RAII 的主要原则是为所有堆分配资源提供所有权，例如动态分配内存或系统对象句柄、析构函数包含要删除或释放资源的代码的堆栈分配对象，以及任何相关清理代码。

## c++智能指针类别

c++ 智能指针主要包括：unique\_ptr,shared\_ptr, weak\_ptr, 这三种，其中auto\_ptr 已被遗弃。

**unique\_ptr**  
只允许基础指针的一个所有者。 可以移到新所有者(具有移动语义)，但不会复制或共享（即我们无法得到指向同一个对象的两个unique\_ptr）。 替换已弃用的 auto\_ptr。 相较于 boost::scoped\_ptr。 unique\_ptr 小巧高效；大小等同于一个指针，支持 rvalue 引用，从而可实现快速插入和对 STL 集合的检索。 头文件：<memory>。

使用unique\_ptr，可以实现以下功能：

1、为动态申请的内存提供异常安全。  
2、将动态申请内存的所有权传递给某个函数。  
3、从某个函数返回动态申请内存的所有权。

4、在容器中保存指针。  
5、所有auto\_ptr应该具有的（但无法在C++ 03中实现的）功能。

如下代码所示：

class A;

// 如果程序执行过程中抛出了异常，unique\_ptr就会释放它所指向的对象

// 传统的new 则不行

unique\_ptr<A> fun1()

{

unique\_ptr p(new A);

//do something

return p;

}

void fun2()

{ // unique\_ptr具有移动语义

unique\_ptr<A> p = f();// 使用移动构造函数

// do something

}// 在函数退出的时候，p以及它所指向的对象都被删除释放

**shared\_ptr**  
采用引用计数的智能指针。 shared\_ptr基于“引用计数”模型实现，多个shared\_ptr可指向同一个动态对象，并维护了一个共享的引用计数器，记录了引用同一对象的shared\_ptr实例的数量。当最后一个指向动态对象的shared\_ptr销毁时，会自动销毁其所指对象(通过delete操作符)。shared\_ptr的默认能力是管理动态内存，但支持自定义的Deleter以实现个性化的资源释放动作。头文件：<memory>。

基本操作：shared\_ptr的创建、拷贝、绑定对象的变更(reset)、shared\_ptr的销毁(手动赋值为nullptr或离开作用域)、指定deleter等操作。

 shared\_ptr的创建,有两种方式，一，使用函数make\_shared(会根据传递的参数调用动态对象的构造函数)；二，使用构造函数(可从原生指针、unique\_ptr、另一个shared\_ptr创建)

shared\_ptr<int> p1 = make\_shared<int>(1);// 通过make\_shared函数

shared\_ptr<int> p2(new int(2));// 通过原生指针构造

此外智能指针若为“空“，即不指向任何对象，则为false，否则为true，可作为条件判断。可以通过两种方式指定deleter，一是构造shared\_ptr时，二是使用reset方法时。可以重载的operator->, operator \*，以及其他辅助操作如unique()、use\_count(), get()等成员方法。

**weak\_ptr**  
结合 shared\_ptr 使用的特例智能指针。 weak\_ptr 提供对一个或多个 shared\_ptr 实例所属对象的访问，但是，不参与引用计数。 如果您想要观察对象但不需要其保持活动状态，请使用该实例。 在某些情况下需要断开 shared\_ptr 实例间的循环引用。 头文件：<memory>。

weak\_ptr的用法如下：

weak\_ptr用于配合shared\_ptr使用，并不影响动态对象的生命周期，即其存在与否并不影响对象的引用计数器。weak\_ptr并没有重载operator->和operator \*操作符，因此不可直接通过weak\_ptr使用对象。提供了expired()与lock()成员函数，前者用于判断weak\_ptr指向的对象是否已被销毁，后者返回其所指对象的shared\_ptr智能指针(对象销毁时返回”空“shared\_ptr)。循环引用的场景：如二叉树中父节点与子节点的循环引用，容器与元素之间的循环引用等。

## 智能指针的循环引用

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**

代码如下：

#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;

}

**需要知道的**：

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。

# [Qt 智能指针学习](http://blog.csdn.net/dbzhang800/article/details/6403285)

标签： [qt](http://www.csdn.net/tag/qt)[delete](http://www.csdn.net/tag/delete)[object](http://www.csdn.net/tag/object)[pointers](http://www.csdn.net/tag/pointers)[reference](http://www.csdn.net/tag/reference)[destructor](http://www.csdn.net/tag/destructor)

2016-11-25 11:38 23737人阅读 [评论](http://blog.csdn.net/dbzhang800/article/details/6403285#comments)(14) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/dbzhang800/article/details/6403285#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

Qt（121） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg C/C++（52） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/dbzhang800/article/details/6403285)

## 从内存泄露开始？

很简单的入门程序，应该比较熟悉吧 ^\_^

#include <QApplication>

#include <QLabel>

int main(int argc, char \*argv[])

{

QApplication app(argc, argv);

QLabel \*label = new QLabel("Hello Dbzhang800!");

label->show();

return app.exec();

}

在  [从 Qt 的 delete 说开来](http://blog.csdn.net/dbzhang800/archive/2011/04/03/6300025.aspx) 一文中，我们提到这个程序存在内存泄露(表现就是析构函数不被调用)，而且当时给出了三种解决方法：

* 将label对象分配到stack而不是heap中
* 给label设置标记位Qt::WA\_DeleteOnClose
* 自己调用delete来删除通过new分配到heap中的 label 对象

注：

* 为了能清楚观察构造和析构函数的调用，我们可以简单子类化了一下QLabel，然后用Label取代前面的QLabel
* class Label :public QLabel
* {
* public:
* Label(const QString& text, QWidget \*parent=NULL)
* :QLabel(text, parent){qDebug("from constructor");}
* ~Label(){qDebug("from destructor");}

};

本文中，我们从智能指针(smart pointer)角度继续考虑这个问题

## 智能指针

为了管理内存等资源，C++程序员通常采用RAII(Resource Acquisition Is Initialization)机制：在类的构造函数中申请资源，然后使用，最后在析构函数中释放资源。

如果没有智能指针，程序员必须保证new对象能在正确的时机delete，四处编写异常捕获代码以释放资源，而智能指针则可以在退出作用域时(不管是正常流程离开或是因异常离开)总调用delete来析构在堆上动态分配的对象。

我们看看Qt家族的智能指针：

|  |  |  |
| --- | --- | --- |
| **智能指针** |  | **引入** |
| QPointer | Qt Object 模型的特性(之一) 注意：析构时不会delete它管理的资源 |  |
| QSharedPointer | 带引用计数 | Qt4.5 |
| QWeakPointer |  | Qt4.5 |
| QScopedPointer |  | Qt4.6 |
| QScopedArrayPointer | QScopedPointer的派生类 | Qt4.6 |
| QSharedDataPointer | 用来实现Qt的隐式共享(Implicit Sharing) | Qt4.0 |
| QExplicitlySharedDataPointer | 显式共享 | Qt4.4 |
|  |  |  |
| std::auto\_ptr |  |  |
| std::shared\_ptr | std::tr1::shared\_ptr | [**C++**](http://lib.csdn.net/base/cplusplus)0x |
| std::weak\_ptr | std::tr1::weak\_ptr | C++0x |
| std::unique\_ptr | boost::scoped\_ptr | C++0x |

注：

* MSVC2010 和 GCC g++ 4.3 支持 C++0x
* MSVC2008 sp1 及 GCC g++ 4.0 支持 tr1

有了这些东西，我们就可以很容易改造我们前面的例子了(只需改变一行)：

std::auto\_ptr<QLabel> label(new QLabel("Hello Dbzhang800!"));

根据你所用的Qt的版本，以及C++编译器的支持程度，你可以选用：

* QScopedPointer
* std::unique\_ptr
* QSharedPointer
* std::shared\_ptr
  + std::tr1::shared\_ptr

## QPointer

如何翻译呢？我不太清楚，保留英文吧。

* The QPointer class is a template class that provides  **guarded pointers**   to QObjects.
* 使用：一个guarded指针，QPointer<T> ，行为和常规的指针 T \* 类似
* 特点：当其指向的对象(T必须是QObject及其派生类)被销毁时，它会被自动置NULL.
  + 注意：它本身析构时不会自动销毁所guarded的对象
* 用途：当你需要保存其他人所拥有的QObject对象的指针时，这点非常有用

一个例子

QPointer<QLabel> label = new QLabel;

label->setText("&Status:");

...

if (label)

label->show();

如果在...部分你将该对象delete掉了，label会自动置NULL，而不会是一个悬挂(dangling)的野指针。

QPointer 属于Qt Object模型的核心机制之一，请注意和其他智能指针的区别。

## std::auto\_ptr

这个没多少要说的。

* 不能让多个auto\_ptr 指向同一个对象！(auto\_ptr被销毁时会自动删除它指向的对象，这样对象会被删除多次)
  + 通过拷贝构造或赋值进行操作时，被拷贝的会自动变成NULL，复制所得的指针将获得资源的唯一所有权
* 智能指针不能指向数组(因为其实现中调用的是delete而非delete[])
* 智能指针不能作为容器类的元素。

在C++0x中，auto\_ptr已经不建议使用，以后应该会被其他3个智能指针所取代。

## QScopedPointer 与 std::unique\_ptr

它们概念上应该是是一样的。下面不再区分：

这是一个很类似auto\_ptr的智能指针，它包装了new操作符在堆上分配的动态对象，能够保证动态创建的对象在任何时候都可以被正确地删除。但它的所有权更加严格，不能转让，一旦获取了对象的管理权，你就无法再从它那里取回来。

无论是QScopedPointer 还是 std::unique\_ptr 都拥有一个很好的名字，它向代码的阅读者传递了明确的信息：这个智能指针只能在本作用域里使用，不希望被转让。因为它的拷贝构造和赋值操作都是私有的，这点我们可以对比QObject及其派生类的对象哈。

**用法** (来自Qt的manual)：

考虑没有智能指针的情况，

void myFunction(bool useSubClass)

{

MyClass \*p = useSubClass ? new MyClass() : new MySubClass;

QIODevice \*device = handsOverOwnership();

if (m\_value > 3) {

delete p;

delete device;

return;

}

try {

process(device);

}

catch (...) {

delete p;

delete device;

throw;

}

delete p;

delete device;

}

我们在异常处理语句中多次书写delete语句，稍有不慎就会导致资源泄露。采用智能指针后，我们就可以将这些异常处理语句简化了：

void myFunction(bool useSubClass)

{

QScopedPointer<MyClass> p(useSubClass ? new MyClass() : new MySubClass);

QScopedPointer<QIODevice> device(handsOverOwnership());

if (m\_value > 3)

return;

process(device);

}

另，我们一开始的例子，也是使用这两个指针的最佳场合了(出main函数作用域就将其指向的对象销毁)。

注意：因为拷贝构造和赋值操作私有的，它也具有auto\_ptr同样的“缺陷”——不能用作容器的元素。

## QSharedPointer 与 std::shared\_ptr

QSharedPointer 与 std::shared\_ptr 行为最接近原始指针，是最像指针的"智能指针"，应用范围比前面的提到的更广。

QSharedPointer 与 QScopedPointer 一样包装了new操作符在堆上分配的动态对象，但它实现的是引用计数型的智能指针 ，可以被自由地拷贝和赋值，在任意的地方共享它，当没有代码使用（引用计数为0）它时才删除被包装的动态分配的对象。shared\_ptr也可以安全地放到标准容器中，并弥补了std::auto\_ptr 和 QScopedPointer 因为转移语义而不能把指针作为容器元素的缺陷。

## QWeakPointer 与 std::weak\_ptr

强引用类型的QSharedPointer已经非常好用，为什么还要有弱引用的 QWeakPointer？

QWeakPointer 是为配合 QSharedPointer 而引入的一种智能指针，它更像是 QSharedPointer 的一个助手(因为它不具有普通指针的行为，没有重载operator\*和->)。它的最大作用在于协助 QSharedPointer 工作，像一个旁观者一样来观测资源的使用情况。

* weak\_ptr 主要是为了避免强引用形成环状。摘自msdn中一段话：
  + A cycle occurs when two or more resources controlled by shared\_ptr objects hold mutually referencing shared\_ptr objects. For example, a circular linked list with three elements has a head node N0; that node holds a shared\_ptr object that owns the next node, N1; that node holds a shared\_ptr object that owns the next node, N2; that node, in turn, holds a shared\_ptr object that owns the head node, N0, closing the cycle. In this situation, none of the reference counts will ever become zero, and the nodes in the cycle will not be freed. To eliminate the cycle, the last node N2 should hold a weak\_ptr object pointing to N0 instead of a shared\_ptr object. Since the weak\_ptr object does not own N0 it doesn't affect N0's reference count, and when the program's last reference to the head node is destroyed the nodes in the list will also be destroyed.
* 在Qt中，对于QObject及其派生类对象，QWeakPointer有特殊处理。它可以作为QPointer的替代品
  + 这种情况下，不需要QSharedPointer的存在
  + 效率比 QPointer 高

## QSharedDataPointer

这是为配合 QSharedData 实现隐式共享(写时复制 copy-on-write))而提供的便利工具。

Qt中众多的类都使用了隐式共享技术，比如QPixmap、QByteArray、QString、...。而我们为自己的类实现隐式共享也很简单，比如要实现一个 Employee类：

* 定义一个只含有一个数据成员(QSharedDataPointer<EmployeeData>) 的 Employee 类
* 我们需要的所有数据成员放置于 派生自QSharedData的 EmployeeData类中。

具体实现看 QSharedDataPointer 的Manual，此处略

## QExplicitlySharedDataPointer

这是为配合 QSharedData 实现显式共享而提供的便利工具。

QExplicitlySharedDataPointer 和 QSharedDataPointer 非常类似，但是它禁用了写时复制功能。这使得我们创建的对象更像一个指针。

一个例子，接前面的Employee：

#include "employee.h"

int main()

{

Employee e1(1001, "Albrecht Durer");

Employee e2 = e1;

e1.setName("Hans Holbein");

}

写时复制技术导致：e1和e2有相同的工号，但有不同名字。与我们期待的不同，显式共享可以解决这个问题，这也使得Employee本身更像一个指针。

## 补遗

先前竟未注意到官方的这两篇文章（这是失败）:

* [http://labs.qt.nokia.com/2009/08/25/count-with-me-how-many-smart-pointer-classes-does-qt-have/](http://labs.qt.nokia.com/2009/08/21/introducing-qscopedpointer/)
* <http://labs.qt.nokia.com/2009/08/21/introducing-qscopedpointer/>

便看看google编码规范中对3个智能指针的建议：

scoped\_ptr

Straightforward and risk-free. Use wherever appropriate.

auto\_ptr

Confusing and bug-prone ownership-transfer semantics. Do not use.

shared\_ptr

Safe with const referents (i.e. shared\_ptr<const T> ). Reference-counted pointers with non-const referents can occasionally be the best design, but try to rewrite with single owners where possible.

## 参考

* Qt manual
* [Smart Pointers FAQ](http://www.csc.lsu.edu/~kosmas/smartPointers.html)
* [The New C++:Smart(er) Pointers](http://drdobbs.com/184403837/)
* <http://en.wikipedia.org/wiki/Smart_pointer>
* [C++:智能指针-TR1的shared\_ptr和weak\_ptr使用介绍](http://zhangjunhd.blog.51cto.com/113473/148628/)
* [【C++Boost】智能指针的标准之争：Boost vs. Loki](http://dozb.bokee.com/1976635.html)
* <http://www2.research.att.com/~bs/C++0xFAQ.html>
* <http://msdn.microsoft.com/zh-cn/library/bb982126.aspx>
* <http://www.codesynthesis.com/~boris/blog/2010/05/24/smart-pointers-in-boost-tr1-cxx-x0/>
* <http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml>

# 进程间通信机制

管道、命名管道、消息队列、共享内存、信号量、信号、socket、读写锁、自旋锁、