<http://www.cnblogs.com/shawnChi/p/5742033.html>

[GeekBand]C++高级编程技术（2）

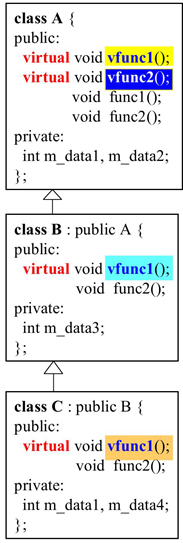
本篇笔记主要分为两个主要部分，第一部分关于对象模型，第二部分是关于new和delete的更加深入的学习。

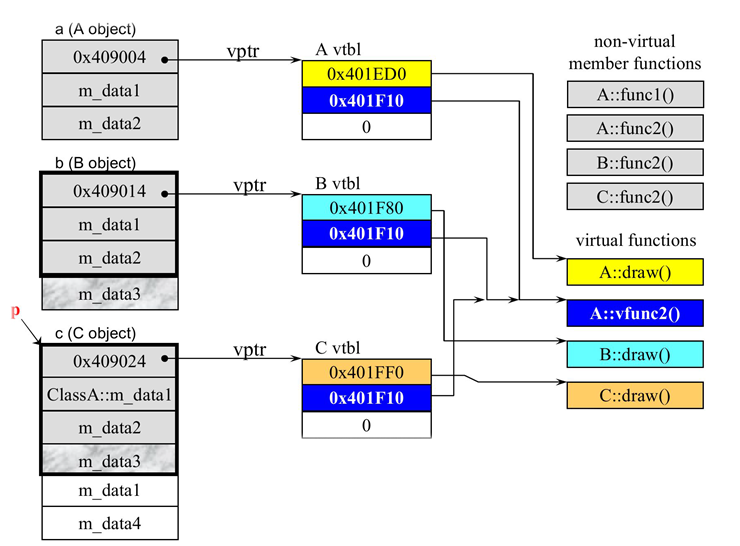
一、对象模型

关于vptr（虚指针）和vtbl（虚函数表）

只要用到了虚函数，对象中就会多一个指向虚函数表的虚指针。在32位环境下，将占4Bytes的空间。

在vtbl中，每一项都是指向自己类应当调用的虚函数的函数指针。





这里提一下，如果父类定义成虚函数,子类中和父类虚函数相同名称,参数表相同的函数会自动变成虚函数。不管加没有virtual关键字。通常我们还是要加上关键字来确保代码可读性。

2．静态绑定与动态绑定

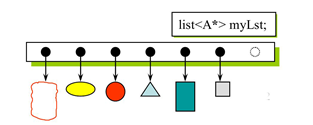
在C中，对于不同的函数名采用静态绑定的方法，每个函数直接对应了一个地址，存储在相应的位置中。在C++中，非虚的成员函数也用静态绑定的方式被存储。如上图中的A::func1()等成员函数。

不过对于虚函数，C++中采用了动态绑定的方法。在上图中，每个虚函数都存储在虚函数表中。当调用虚函数时，编译器会随着上图中的路径找到正确的函数调用。

由于动态绑定，不管什么地方调用虚函数，总能得到正确的结果这个机制限制了虚函数应该被虚函数覆盖。

虚函数可以由以下的两种方式得到。





关于类的对象模型的内存分配，涉及到对象的位对齐的规则，在博客中的另一篇文章中有简单介绍。

二、再谈new和delete

定位new和delete运算符及其重载

new和delete在使用时可以有一个可选的指针类型的参数，用来指定内存分配的起始地址。如果没有这一参数，则会在堆空间中自动分配一段合适大小的空间。

默认的一个定位new函数是：

void\* operator new(size\_t size,void \*start)

在使用时可以采用例如如下的方法：int \*p = new(0x12345678) int;

事实上，我们还可以使用别的参数列进行new操作，我们也可以对operator new和operator delete进行重载。例如下面的两个常用的参数列：

void\* operator new(size\_t size,long extra)//extra参数用于多申请一段存储空间，专门用来存储一些特别的信息，例如引用计数的信息。

void\* operator new(size\_t size,long extra,char init)

new和delete操作固有的定义是：

inline void \* operator new(size\_t size){return malloc(size);}

inline void \* operator new[](size\_t size){return malloc(size);}//这里在调用时size会自动进行计算。

inline void operator delete(void \* ptr){free(ptr);}

inline void operator delete[](void \* ptr){free(ptr);}

注意，如果需要进行重载，new第一个参数始终应该是size\_t格式的，delete的第一个参数始终是void\*类型。此外，通常情况下有参数的delete不会被调用，而是继续采用无参的方法，直接释放内存。

此外，new和delete也可以作为类的成员函数重载，这样重载出来的操作将仅用于这个类。需要注意的是，new和delete操作应该是静态的，其原因在于：

1.创建对象时，先分配内存，此时没有对象，只能是静态的。

2.删除对象，先执行析构，析构后已经没有对象了，所以这里也只能是静态的。

只有当构造函数失败时，才会去寻找匹配的operator delete函数去释放空间；如果没有定义相应的delete函数，就代表放弃处理创建失败的这一异常。

注：

当一对operator new 和operator delete除了第一个参数之外，剩下的参数都一致时，称这两个操作"匹配"。

从new操作显式抛出异常，并不会触发特殊的delete。从new操作中抛出异常，代表内存分配没有进行，因此也就不需要释放内存；只有再分配内存之后，构造时产生异常才会触发特殊的delete。

下面是关于new、delete操作的测试代码：

#ifndef HEADER\_H\_INCLUDED

#define HEADER\_H\_INCLUDED

#include <iostream>

#include <stdlib.h>

using namespace std;

class Fruit{

int no;

double weight;

char key;

public:

void print() { cout<<"no =\t"<<no<<"\tweight =\t"<<weight<<"\tkey =\t"<<key<<endl; }

virtual void process(){ }

Fruit(int n = 0,double w = 0,char k = 0):no(n),weight(w),key(k){ cout<<"Ctor of Fruit is called.\nno =\t"<<no<<"\tweight =\t"<<weight<<"\tkey =\t"<<key<<endl;}

virtual ~Fruit(){cout<<"Dtor of Fruit is called.\nno =\t"<<no<<endl;}

};

class Apple: public Fruit{

int size;

char type;

public:

void save() { }

virtual void process(){ }

Apple(int s = 0,char t = 0,int n = 0,double w = 0,char k = 0):Fruit(n,w,k),size(s),type(t){

cout<<"Ctor of Apple is called.\n";

print();

cout<< "size =\t"<<size<<"\ttype = \t"<<type<<endl;

//注：下面这条语句是为了测试重载版本的delete而特别添加的.

if(s<0) throw 1;

}

virtual ~Apple(){

cout<<"Dtor of Apple is called."<<endl;

}

static void \* operator new(size\_t size);

static void \* operator new(size\_t size,long extra);

static void operator delete(void \* ptr);

static void operator delete(void\* ptr,long extra);

};

void\* operator new(size\_t size)

{

cout<<"::operatornew(size\_t) is called.\n";

return malloc(size);

}

void operator delete(void\* ptr)

{

cout<<"::operatordelete(void\*) is called.\n";

free(ptr);

}

#endif // HEADER\_H\_INCLUDED

头文件

#include "header.h"

using namespace std;

void\* Apple::operator new(size\_t size)

{

cout<<"void\* Apple::operator new(size\_t) is called.\n";

return malloc(size);

}

void\* Apple::operator new(size\_t size,long extra)

{

cout<<"void\* Apple::operator new(size\_t,long) is called.\n";

return malloc(size+extra);

}

void Apple::operator delete(void\* ptr)

{

cout<<"void Apple::operator delete(void\*) is called.\n";

free(ptr);

}

void Apple::operator delete(void\* ptr,long extra)

{

cout<<"void Apple::operator delete(void\*,long) is called.\n";

free(ptr);

}

int main()

{

cout<<"======Fruit new,delete test=======\n";

//这部分的结果体现了：

//1.在没有为类指定单独的new和delete时，会调用全局的new和delete；

//2.造函数先分配内存，再进行构造的特点和析构函数先进行析构，再释放内存的特点；

Fruit \* \_Fruit = new Fruit(0,0,'A');

delete \_Fruit;

cout<<"======Apple new,delete test======\n";

//这部分的结果体现了:

//1.继承关系先构造基类，再构造自身

//2.析构时先析构自身，再析构基类

//3.类里边有单独的new和delete时，使用类内定义的new和delete。

Apple \* \_Apple = new Apple(0,0,1,0,'B');

delete \_Apple;

cout<<"===Apple multi-parameters new,delete test===\n";

//这部分的结果体现了：

//1.显式delete操作总是调用delete(void\*),而不管是如何new出来的

\_Apple = new(long(100)) Apple(0,0,2,0,'C');

delete \_Apple;

cout<<"=Apple new,delete failed test=\n";

//这部分的结果体现了：

//1.当ctor失败时，会去寻找匹配的dtor.

Apple \* \_Apple\_2;

try{

\_Apple\_2 = new(long(100)) Apple(-1);

}

catch(...)

{

cout<<"Catch Error.";

}

return 0;

}