Linux：

if [ str1 = str2 ] 当两个串有相同内容、长度时为真

if [ str1 != str2 ] 当串str1和str2不等时为真

if [ -n str1 ] 当串的长度大于0时为真(串非空)

if [ -z str1 ] 当串的长度为0时为真(空串)

if [ str1 ]　 当串str1为非空时为真

gcc工具链

objdump -S objfile

objdump -d objfile

objdump -h objfile

gdb调试

gdb 进程名字 coredump文件名字

gdb attach 进程号

bt ：打印堆栈

wget

wget http://www.linuxde.net/testfile.zip

linux个别常用命令

sed 's/要被取代的字串/新的字串/g'

例如 cat /etc/passwd | sed 's/root:x:/root::/g' > /3rd\_rw/passwd.txt

查看线程占用CPU使用率

1 先用ps + grep找出该死的进程pid，比如 30420

2 top -H -p pid(进程号)(top然后shift+H可以看出某个线程)所有该进程的线程都列出来了。看看哪个线程pid占用最多，然后将这个pid转换为16进制，如 44bf，注意要小写

3 jstack 30420 | less，然后查找 nid=0x44bf，哦，找到了

c++学习：

Overload(重载)：在C++程序中，可以将语义、功能相似的几个函数用同一个名字表示，但参数或返回值不同（包括类型、顺序不同），即函数重载。

（1）相同的范围（在同一个类中）；

（2）函数名字相同；

（3）参数不同；

（4）virtual 关键字可有可无。

Override(覆盖)：是指派生类函数覆盖基类函数，特征是：

（1）不同的范围（分别位于派生类与基类）；

（2）函数名字相同；

（3）参数相同；

（4）基类函数必须有virtual 关键字。

（5）Override函数的访问修饰符可以不同，尽管virtual函数是private的，在派生类中Override的函数可以是public或protected的

（6）覆盖的函数不能是static函数；即virtual关键字不需要和static关键字一起使用

Overwrite(重写)：是指派生类的函数屏蔽了与其同名的基类函数，规则如下：

（1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual关键字，基类的函数将被隐藏（注意别与重载混淆：类的范围不同）。

（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual关键字。此时，基类的函数被隐藏（注意别与覆盖混淆：virtual关键字）

C++中static

• 出现在类体外的函数定义不能指定关键字static；

•虚函数不需要static关键字

• 静态成员之间可以相互访问，包括静态成员函数访问静态数据成员和访问静态成员函数；

• 非静态成员函数可以任意地访问静态成员函数和静态数据成员；

• 静态成员函数不能访问非静态成员函数和非静态数据成员；

• 由于没有this指针的额外开销，因此静态成员函数与类的全局函数相比速度上会有少许的增长；

• 调用静态成员函数，可以用成员访问操作符(.)和(->)为一个类的对象或指向类对象的指针调用静态成员函数，也可以直接使用如下格式：

＜类名＞::＜静态成员函数名＞（＜参数表＞）

•基类定义的static变量，派生类如果没有定义同名的变量，可以访问基类的静态变量，并且修改其内容，基类再查看这个变量的内容会发现内容也被修改。（如果不是static变量，则基类和派生类修改的都是各自变量的内容）

STL（Standard Template Library）之map

1 头文件

#include <map>

2 定义

map<string, int> my\_Map;

或者是typedef map<string, int> MY\_MAP;

MY\_MAP my\_Map;

3 插入数据

(1) my\_Map["a"] = 1;

(2) my\_Map.insert(map<string, int>::value\_type("b",2));

(3) my\_Map.insert(pair<string,int>("c",3));

(4) my\_Map.insert(make\_pair<string,int>("d",4));

4 查找数据和修改数据

(1) int i = my\_Map["a"];

my\_Map["a"] = i;

(2) MY\_MAP::iterator my\_Itr;

my\_Itr.find("b");

int j = my\_Itr->second;

my\_Itr->second = j;

不过注意，键本身是不能被修改的，除非删除。

5 删除数据

(1) my\_Map.erase(my\_Itr);

(2) my\_Map.erase("c");

还是注意，第一种情况在迭代期间是不能被删除的，道理和foreach时不能删除元素一样。

6 迭代数据

for (my\_Itr=my\_Map.begin(); my\_Itr!=my\_Map.end(); ++my\_Itr) {}

7 其它方法

my\_Map.size() 返回元素数目

my\_Map.empty() 判断是否为空

my\_Map.clear() 清空所有元素

C++中指针和引用的区别

指针和引用在C++中很常用，但是对于它们之间的区别很多初学者都不是太熟悉，下面来谈谈他们2者之间的区别和用法。

1.指针和引用的定义和性质区别：

(1)指针：指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名而已。如：

int a=1;int \*p=&a;

int a=1;int &b=a;

上面定义了一个整形变量和一个指针变量p，该指针变量指向a的存储单元，即p的值是a存储单元的地址。

而下面2句定义了一个整形变量a和这个整形a的引用b，事实上a和b是同一个东西，在内存占有同一个存储单元。

(2)可以有const指针，但是没有const引用；

(3)指针可以有多级，但是引用只能是一级（int \*\*p；合法 而 int &&a是不合法的）

(4)指针的值可以为空，但是引用的值不能为NULL，并且引用在定义的时候必须初始化；

(5)指针的值在初始化后可以改变，即指向其它的存储单元，而引用在进行初始化后就不会再改变了。

(6)"sizeof引用"得到的是所指向的变量(对象)的大小，而"sizeof指针"得到的是指针本身的大小；

(7)指针和引用的自增(++)运算意义不一样；

2.指针和引用作为函数参数进行传递时的区别。

(1)指针作为参数进行传递：

#include<iostream>

using namespace std;

void swap(int \*a,int \*b)

{

　　int temp=\*a;

　　\*a=\*b;

　　\*b=temp;

}

int main(void)

{

　　int a=1,b=2;

　　swap(&a,&b);

　　cout<<a<<" "<<b<<endl;

　　system("pause");

　　return 0;

}

结果为2 1；

用指针传递参数，可以实现对实参进行改变的目的，是因为传递过来的是实参的地址，因此使用\*a实际上是取存储实参的内存单元里的数据，即是对实参进行改变，因此可以达到目的。

再看一个程序;

#include<iostream>

using namespace std;

void test(int \*p)

{

　　int a=1;

　　p=&a;

　　cout<<p<<" "<<\*p<<endl;

}

int main(void)

{

int \*p=NULL;

test(p);

if(p==NULL)

cout<<"指针p为NULL"<<endl;

system("pause");

return 0;

}

运行结果为：

0x22ff44 1

指针p为NULL

大家可能会感到奇怪，怎么回事，不是传递的是地址么，怎么p会是NULL？事实上，在main函数中声明了一个指针p，并赋值为NULL，当调用test函数时，事实上传递的也是地址，只不过传递的是指地址。也就是说将指针作为参数进行传递时，事实上也是值传递，只不过传递的是地址。当把指针作为参数进行传递时，也是将实参的一个拷贝传递给形参，即上面程序main函数中的p何test函数中使用的p不是同一个变量，存储2个变量p的单元也不相同（只是2个p指向同一个存储单元），那么在test函数中对p进行修改，并不会影响到main函数中的p的值。

如果要想达到也同时修改的目的的话，就得使用引用了。

（2）将引用作为函数的参数进行传递。

在讲引用作为函数参数进行传递时，实质上传递的是实参本身，即传递进来的不是实参的一个拷贝，因此对形参的修改其实是对实参的修改，所以在用引用进行参数传递时，不仅节约时间，而且可以节约空间。

看下面这个程序：

#include<iostream>

using namespace std;

void test(int &a)

{

　　cout<<&a<<" "<<a<<endl;

}

int main(void)

{

int a=1;

cout<<&a<<" "<<a<<endl;

test(a);

system("pause");

return 0;

}

输出结果为： 0x22ff44 1

0x22ff44 1

再看下这个程序：

这足以说明用引用进行参数传递时，事实上传递的是实参本身，而不是拷贝。

所以在上述要达到同时修改指针的目的的话，就得使用引用了。

#include<iostream>

using namespace std;

void test(int \*&p)

{

　　int a=1;

　　p=&a;

　　cout<<p<<" "<<\*p<<endl;

}

int main(void)

{

int \*p=NULL;

test(p);

if(p!=NULL)

cout<<"指针p不为NULL"<<endl;

system("pause");

return 0;

}

输出结果为：0x22ff44 1

指针p不为NULL

html

(1) 在js代码中更改css（样式）元素的属性

var div=document.getElementById('div');

div.style.setProperty('border','1px red solid');

<marquee behavior=alternate scrollamount =3>aaaa</marquee>

**智能指针**

导读

最近在补看《C++ Primer Plus》第六版，这的确是本好书，其中关于智能指针的章节解析的非常清晰，一解我以前的多处困惑。C++面试过程中，很多面试官都喜欢问智能指针相关的问题，比如你知道哪些智能指针？shared\_ptr的设计原理是什么？如果让你自己设计一个智能指针，你如何完成？等等……。而且在看开源的C++项目时，也能随处看到智能指针的影子。这说明智能指针不仅是面试官爱问的题材，更是非常有实用价值。

下面是我在看智能指针时所做的笔记，希望能够解决你对智能指针的一些困扰。

目录

智能指针背后的设计思想

C++智能指针简单介绍

为什么摒弃auto\_ptr？

unique\_ptr为何优于auto\_ptr？

如何选择智能指针？

正文

1. 智能指针背后的设计思想

我们先来看一个简单的例子：

复制代码

void remodel(std::string & str)

{

std::string \* ps = new std::string(str);

...

if (weird\_thing())

throw exception();

str = \*ps;

delete ps;

return;

}

复制代码

当出现异常时（weird\_thing()返回true），delete将不被执行，因此将导致内存泄露。

如何避免这种问题？有人会说，这还不简单，直接在throw exception();之前加上delete ps;不就行了。是的，你本应如此，问题是很多人都会忘记在适当的地方加上delete语句（连上述代码中最后的那句delete语句也会有很多人忘记吧），如果你要对一个庞大的工程进行review，看是否有这种潜在的内存泄露问题，那就是一场灾难！

这时我们会想：当remodel这样的函数终止（不管是正常终止，还是由于出现了异常而终止），本地变量都将自动从栈内存中删除—因此指针ps占据的内存将被释放，如果ps指向的内存也被自动释放，那该有多好啊。

我们知道析构函数有这个功能。如果ps有一个析构函数，该析构函数将在ps过期时自动释放它指向的内存。但ps的问题在于，它只是一个常规指针，不是有析构凼数的类对象指针。如果它指向的是对象，则可以在对象过期时，让它的析构函数删除指向的内存。

这正是 auto\_ptr、unique\_ptr和shared\_ptr这几个智能指针背后的设计思想。我简单的总结下就是：将基本类型指针封装为类对象指针（这个类肯定是个模板，以适应不同基本类型的需求），并在析构函数里编写delete语句删除指针指向的内存空间。

因此，要转换remodel()函数，应按下面3个步骤进行：

包含头义件memory（智能指针所在的头文件）；

将指向string的指针替换为指向string的智能指针对象；

删除delete语句。

下面是使用auto\_ptr修改该函数的结果：

复制代码

# include <memory>

void remodel (std::string & str)

{

std::auto\_ptr<std::string> ps (new std::string(str))；

...

if (weird\_thing ())

throw exception()；

str = \*ps；

// delete ps； NO LONGER NEEDED

return;

}

复制代码

2. C++智能指针简单介绍

STL一共给我们提供了四种智能指针：auto\_ptr、unique\_ptr、shared\_ptr和weak\_ptr（本文章暂不讨论）。

模板auto\_ptr是C++98提供的解决方案，C+11已将将其摒弃，并提供了另外两种解决方案。然而，虽然auto\_ptr被摒弃，但它已使用了好多年：同时，如果您的编译器不支持其他两种解决力案，auto\_ptr将是唯一的选择。

使用注意点

所有的智能指针类都有一个explicit构造函数，以指针作为参数。比如auto\_ptr的类模板原型为：

templet<class T>

class auto\_ptr {

explicit auto\_ptr(X\* p = 0) ;

...

};

因此不能自动将指针转换为智能指针对象，必须显式调用：

shared\_ptr<double> pd;

double \*p\_reg = new double;

pd = p\_reg; // not allowed (implicit conversion)

pd = shared\_ptr<double>(p\_reg); // allowed (explicit conversion)

shared\_ptr<double> pshared = p\_reg; // not allowed (implicit conversion)

shared\_ptr<double> pshared(p\_reg); // allowed (explicit conversion)

对全部三种智能指针都应避免的一点：

string vacation("I wandered lonely as a cloud.");

shared\_ptr<string> pvac(&vacation); // No

pvac过期时，程序将把delete运算符用于非堆内存，这是错误的。

使用举例

复制代码

#include <iostream>

#include <string>

#include <memory>

class report

{

private:

std::string str;

public:

report(const std::string s) : str(s) {

std::cout << "Object created.\n";

}

~report() {

std::cout << "Object deleted.\n";

}

void comment() const {

std::cout << str << "\n";

}

};

int main() {

{

std::auto\_ptr<report> ps(new report("using auto ptr"));

ps->comment();

}

{

std::shared\_ptr<report> ps(new report("using shared ptr"));

ps->comment();

}

{

std::unique\_ptr<report> ps(new report("using unique ptr"));

ps->comment();

}

return 0;

}

复制代码

3. 为什么摒弃auto\_ptr？

先来看下面的赋值语句:

auto\_ptr< string> ps (new string ("I reigned lonely as a cloud.”）;

auto\_ptr<string> vocation;

vocaticn = ps;

上述赋值语句将完成什么工作呢？如果ps和vocation是常规指针，则两个指针将指向同一个string对象。这是不能接受的，因为程序将试图删除同一个对象两次——一次是ps过期时，另一次是vocation过期时。要避免这种问题，方法有多种：

定义陚值运算符，使之执行深复制。这样两个指针将指向不同的对象，其中的一个对象是另一个对象的副本，缺点是浪费空间，所以智能指针都未采用此方案。

建立所有权（ownership）概念。对于特定的对象，只能有一个智能指针可拥有，这样只有拥有对象的智能指针的构造函数会删除该对象。然后让赋值操作转让所有权。这就是用于auto\_ptr和uniqiie\_ptr 的策略，但unique\_ptr的策略更严格。

创建智能更高的指针，跟踪引用特定对象的智能指针数。这称为引用计数。例如，赋值时，计数将加1，而指针过期时，计数将减1,。当减为0时才调用delete。这是shared\_ptr采用的策略。

当然，同样的策略也适用于复制构造函数。

每种方法都有其用途，但为何说要摒弃auto\_ptr呢？

下面举个例子来说明。

复制代码

#include <iostream>

#include <string>

#include <memory>

using namespace std;

int main() {

auto\_ptr<string> films[5] =

{

auto\_ptr<string> (new string("Fowl Balls")),

auto\_ptr<string> (new string("Duck Walks")),

auto\_ptr<string> (new string("Chicken Runs")),

auto\_ptr<string> (new string("Turkey Errors")),

auto\_ptr<string> (new string("Goose Eggs"))

};

auto\_ptr<string> pwin;

pwin = films[2]; // films[2] loses ownership. 将所有权从films[2]转让给pwin，此时films[2]不再引用该字符串从而变成空指针

cout << "The nominees for best avian baseballl film are\n";

for(int i = 0; i < 5; ++i)

cout << \*films[i] << endl;

cout << "The winner is " << \*pwin << endl;

cin.get();

return 0;

}

复制代码

运行下发现程序崩溃了，原因在上面注释已经说的很清楚，films[2]已经是空指针了，下面输出访问空指针当然会崩溃了。但这里如果把auto\_ptr换成shared\_ptr或unique\_ptr后，程序就不会崩溃，原因如下：

使用shared\_ptr时运行正常，因为shared\_ptr采用引用计数，pwin和films[2]都指向同一块内存，在释放空间时因为事先要判断引用计数值的大小因此不会出现多次删除一个对象的错误。

使用unique\_ptr时编译出错，与auto\_ptr一样，unique\_ptr也采用所有权模型，但在使用unique\_ptr时，程序不会等到运行阶段崩溃，而在编译器因下述代码行出现错误：

unique\_ptr<string> pwin;

pwin = films[2]; // films[2] loses ownership.

指导你发现潜在的内存错误。

这就是为何要摒弃auto\_ptr的原因，一句话总结就是：避免潜在的内存崩溃问题。

4. unique\_ptr为何优于auto\_ptr？

可能大家认为前面的例子已经说明了unique\_ptr为何优于auto\_ptr，也就是安全问题，下面再叙述的清晰一点。

请看下面的语句:

auto\_ptr<string> p1(new string ("auto") ； //#1

auto\_ptr<string> p2; //#2

p2 = p1; //#3

在语句#3中，p2接管string对象的所有权后，p1的所有权将被剥夺。前面说过，这是好事，可防止p1和p2的析构函数试图刪同—个对象；

但如果程序随后试图使用p1，这将是件坏事，因为p1不再指向有效的数据。

下面来看使用unique\_ptr的情况：

unique\_ptr<string> p3 (new string ("auto"); //#4

unique\_ptr<string> p4； //#5

p4 = p3; //#6

编译器认为语句#6非法，避免了p3不再指向有效数据的问题。因此，unique\_ptr比auto\_ptr更安全。

但unique\_ptr还有更聪明的地方。

有时候，会将一个智能指针赋给另一个并不会留下危险的悬挂指针。假设有如下函数定义：

unique\_ptr<string> demo(const char \* s)

{

unique\_ptr<string> temp (new string (s))；

return temp；

}

并假设编写了如下代码：

unique\_ptr<string> ps;

ps = demo('Uniquely special")；

demo()返回一个临时unique\_ptr，然后ps接管了原本归返回的unique\_ptr所有的对象，而返回时临时的 unique\_ptr 被销毁，也就是说没有机会使用 unique\_ptr 来访问无效的数据，换句话来说，这种赋值是不会出现任何问题的，即没有理由禁止这种赋值。实际上，编译器确实允许这种赋值，这正是unique\_ptr更聪明的地方。

总之，当程序试图将一个 unique\_ptr 赋值给另一个时，如果源 unique\_ptr 是个临时右值，编译器允许这么做；如果源 unique\_ptr 将存在一段时间，编译器将禁止这么做，比如：

unique\_ptr<string> pu1(new string ("hello world"));

unique\_ptr<string> pu2;

pu2 = pu1; // #1 not allowed

unique\_ptr<string> pu3;

pu3 = unique\_ptr<string>(new string ("You")); // #2 allowed

其中#1留下悬挂的unique\_ptr(pu1)，这可能导致危害。而#2不会留下悬挂的unique\_ptr，因为它调用 unique\_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而已的行为表明，unique\_ptr 优于允许两种赋值的auto\_ptr 。

当然，您可能确实想执行类似于#1的操作，仅当以非智能的方式使用摒弃的智能指针时（如解除引用时），这种赋值才不安全。要安全的重用这种指针，可给它赋新值。C++有一个标准库函数std::move()，让你能够将一个unique\_ptr赋给另一个。下面是一个使用前述demo()函数的例子，该函数返回一个unique\_ptr<string>对象：

使用move后，原来的指针仍转让所有权变成空指针，可以对其重新赋值。

unique\_ptr<string> ps1, ps2;

ps1 = demo("hello");

ps2 = move(ps1);

ps1 = demo("alexia");

cout << \*ps2 << \*ps1 << endl;

5. 如何选择智能指针？

在掌握了这几种智能指针后，大家可能会想另一个问题：在实际应用中，应使用哪种智能指针呢？

下面给出几个使用指南。

（1）如果程序要使用多个指向同一个对象的指针，应选择shared\_ptr。这样的情况包括：

有一个指针数组，并使用一些辅助指针来标示特定的元素，如最大的元素和最小的元素；

两个对象包含都指向第三个对象的指针；

STL容器包含指针。很多STL算法都支持复制和赋值操作，这些操作可用于shared\_ptr，但不能用于unique\_ptr（编译器发出warning）和auto\_ptr（行为不确定）。如果你的编译器没有提供shared\_ptr，可使用Boost库提供的shared\_ptr。

（2）如果程序不需要多个指向同一个对象的指针，则可使用unique\_ptr。如果函数使用new分配内存，并返还指向该内存的指针，将其返回类型声明为unique\_ptr是不错的选择。这样，所有权转让给接受返回值的unique\_ptr，而该智能指针将负责调用delete。可将unique\_ptr存储到STL容器在那个，只要不调用将一个unique\_ptr复制或赋给另一个算法（如sort()）。例如，可在程序中使用类似于下面的代码段。

复制代码

unique\_ptr<int> make\_int(int n)

{

return unique\_ptr<int>(new int(n));

}

void show(unique\_ptr<int> &p1)

{

cout << \*a << ' ';

}

int main()

{

...

vector<unique\_ptr<int> > vp(size);

for(int i = 0; i < vp.size(); i++)

vp[i] = make\_int(rand() % 1000); // copy temporary unique\_ptr

vp.push\_back(make\_int(rand() % 1000)); // ok because arg is temporary

for\_each(vp.begin(), vp.end(), show); // use for\_each()

...

}

复制代码

其中push\_back调用没有问题，因为它返回一个临时unique\_ptr，该unique\_ptr被赋给vp中的一个unique\_ptr。另外，如果按值而不是按引用给show()传递对象，for\_each()将非法，因为这将导致使用一个来自vp的非临时unique\_ptr初始化pi，而这是不允许的。前面说过，编译器将发现错误使用unique\_ptr的企图。

在unique\_ptr为右值时，可将其赋给shared\_ptr，这与将一个unique\_ptr赋给一个需要满足的条件相同。与前面一样，在下面的代码中，make\_int()的返回类型为unique\_ptr<int>：

unique\_ptr<int> pup(make\_int(rand() % 1000)); // ok

shared\_ptr<int> spp(pup); // not allowed, pup as lvalue

shared\_ptr<int> spr(make\_int(rand() % 1000)); // ok

模板shared\_ptr包含一个显式构造函数，可用于将右值unique\_ptr转换为shared\_ptr。shared\_ptr将接管原来归unique\_ptr所有的对象。

在满足unique\_ptr要求的条件时，也可使用auto\_ptr，但unique\_ptr是更好的选择。如果你的编译器没有unique\_ptr，可考虑使用Boost库提供的scoped\_ptr，它与unique\_ptr类似。

c++深拷贝和浅拷贝

当用一个已初始化过了的自定义类类型对象去初始化另一个新构造的对象的时候，拷贝构造函数就会被自动调用。也就是说，当类的对象需要拷贝时，拷贝构造函数将会被调用。以下情况都会调用拷贝构造函数：

（1）一个对象以值传递的方式传入函数体

（2）一个对象以值传递的方式从函数返回

（3）一个对象需要通过另外一个对象进行初始化。

如果在类中没有显式地声明一个拷贝构造函数，那么，编译器将会自动生成一个默认的拷贝构造函数，该构造函数完成对象之间的位拷贝。位拷贝又称浅拷贝，后面将进行说明。

自定义拷贝构造函数是一种良好的编程风格，它可以阻止编译器形成默认的拷贝构造函数，提高源码效率。

浅拷贝和深拷贝

　　在某些状况下，类内成员变量需要动态开辟堆内存，如果实行位拷贝，也就是把对象里的值完全复制给另一个对象，如A=B。这时，如果B中有一个成员变量指针已经申请了内存，那A中的那个成员变量也指向同一块内存。这就出现了问题：当B把内存释放了（如：析构），这时A内的指针就是野指针了，出现运行错误。

　　深拷贝和浅拷贝可以简单理解为：如果一个类拥有资源，当这个类的对象发生复制过程的时候，资源重新分配，这个过程就是深拷贝，反之，没有重新分配资源，就是浅拷贝。下面举个深拷贝的例子。

#include <iostream>

using namespace std;

class CA

{

　public:

　　CA(int b,char\* cstr)

　　{

　　　a=b;

　　　str=new char[b];

　　　strcpy(str,cstr);

　　}

　　CA(const CA& C)

　　{

　　　a=C.a;

　　　str=new char[a];

//深拷贝

　　　if(str!=0)

　　　　strcpy(str,C.str);

　　}

　　void Show()

　　{

　　　cout<<str<<endl;

　　}

　　~CA()

　　{

　　　delete str;

　　}

　private:

　　int a;

　　char \*str;

};

int main()

{

　CA A(10,"Hello!");

　CA B=A;

　B.Show();

　return 0;

}

来总结一下关于 深拷贝与浅拷贝需要知道的基本概念和知识：

（1）什么时候用到拷贝函数？

a.一个对象以值传递的方式传入函数体；

b.一个对象以值传递的方式从函数返回；

c.一个对象需要通过另外一个对象进行初始化。

如果在类中没有显式地声明一个拷贝构造函数，那么，编译器将会自动生成一个默认的拷贝构造函数，该构造函数完成对象之间的位拷贝。位拷贝又称浅拷贝

（2）是否应该自定义拷贝函数？

（3）什么叫深拷贝？什么是浅拷贝？两者异同？

自定义拷贝构造函数是一种良好的编程风格，它可以阻止编译器形成默认的拷贝构造函数，提高源码效率。

如果一个类拥有资源，当这个类的对象发生复制过程的时候，资源重新分配，这个过程就是深拷贝，反之，没有重新分配资源，就是浅拷贝。

（4）深拷贝好还是浅拷贝好？

如果实行位拷贝，也就是把对象里的值完全复制给另一个对象，如A=B。这时，如果B中有一个成员变量指针已经申请了内存，那A中的那个成员变量也指向同一块内存。这就出现了问题：当B把内存释放了（如：析构），这时A内的指针就是野指针了，出现运行错误。

**C++模板**

　　模板是C++支持**参数化**多态的工具，使用模板可以使用户为类或者函数声明一种一般模式，使得类中的某些数据成员或者成员函数的参数、返回值取得任意类型。

　　模板是一种对**类型**进行**参数化**的工具；

　　通常有两种形式：**函数模板**和**类模板**；

　　函数模板针对仅**参数类型**不同的**函数**；

　　类模板针对仅**数据成员**和**成员函数类型**不同的类。

**使用模板的目的就是能够让程序员编写与类型无关的代码。**比如编写了一个交换两个整型int 类型的swap函数，这个函数就只能实现**int** 型，对**double**，字符这些类型无法实现，要实现这些类型的交换就要重新编写另一个**swap**函数。使用模板的目的就是要让这程序的实现与类型无关，比如一个**swap**模板函数，即可以实现**int** 型，又可以实现double型的交换。模板可以应用于函数和类。下面分别介绍。

**注意：模板的声明或定义只能在全局，命名空间或类范围内进行。即不能在局部范围，函数内进行，比如不能在main函数中声明或定义一个模板。**

**一、函数模板通式**

**1、函数模板的格式：template <class 形参名，class 形参名，......> 返回类型 函数名(参数列表)**

**{**

**函数体**

**}**

　　其中**template**和**class**是关见字，**class**可以用**typename** 关见字代替，在这里**typename 和class没区别**，<>括号中的参数叫**模板形参**，模板形参和函数形参很相像，**模板形参不能为空。一但声明了模板函数就可以用模板函数的形参名声明类中的成员变量和成员函数，即可以在该函数中使用内置类型的地方都可以使用模板形参名。**模板形参需要调用该模板函数时提供的模板实参来初始化模板形参，一旦编译器确定了实际的模板实参类型就称他实例化了函数模板的一个实例。比如**swap**的模板函数形式为

**template <class T> void swap(T& a, T& b){}，**

当调用这样的模板函数时类型T就会被被调用时的类型所代替，比如**swap(a,b)**其中**a**和**b**是**int** 型，这时模板函数swap中的形参**T**就会被**int** 所代替，模板函数就变为**swap(int &a, int &b)**。而当**swap(c,d)**其中**c**和**d**是**double**类型时，模板函数会被替换为**swap(double &a, double &b)**，这样就实现了函数的实现与类型无关的代码。

　　2、注意：对于函数模板而言不存在 **h(int,int)** 这样的调用，**不能在函数调用的参数中指定模板形参的类型，对函数模板的调用应使用实参推演来进行**，即只能进行**h(2,3)** 这样的调用，或者**int a, b; h(a,b)**。

**函数模板的示例演示将在下文中涉及！**

**二、类模板通式**

　　1、类模板的格式为：

**template<class  形参名，class 形参名，…>   class 类名**

**{ ... };**

　　类模板和函数模板都是以**template**开始后接模板形参列表组成，模板形参不能为空，**一但声明了类模板就可以用类模板的形参名声明类中的成员变量和成员函数，即可以在类中使用内置类型的地方都可以使用模板形参名来声明。**比如

**template<class T> class A{public: T a; T b; T hy(T c, T &d);};**

在类**A**中声明了两个类型为**T**的成员变量**a**和**b**，还声明了一个返回类型为**T**带两个参数类型为**T**的函数**hy**。

　　2、类模板对象的创建：比如一个模板类**A**，则使用类模板创建对象的方法为**A<int> m;**在类**A**后面跟上一个**<>**尖括号并在里面填上相应的类型，这样的话类**A**中凡是用到模板形参的地方都会被**int** 所代替。当类模板有两个模板形参时创建对象的方法为**A<int, double> m;**类型之间用逗号隔开。

　　3、对于类模板，模板形参的类型必须在类名后的尖括号中明确指定。比如**A<2> m;**用这种方法把模板形参设置为**int**是错误的（**编译错误：error C2079: 'a' uses undefined class 'A<int>'**），**类模板形参不存在实参推演的问题。**也就是说不能把整型值**2**推演为**int** 型传递给模板形参。要把类模板形参调置为**int** 型必须这样指定**A<int> m**。

　　4、在类模板外部定义成员函数的方法为：

**template<模板形参列表> 函数返回类型 类名<模板形参名>::函数名(参数列表){函数体}，**

比如有两个模板形参**T1**，**T2**的类**A**中含有一个**void h()**函数，则定义该函数的语法为：

**template<class T1,class T2> void A<T1,T2>::h(){}。**

注意：当在类外面定义类的成员时**template**后面的模板形参应与要定义的类的模板形参一致。

　　5、再次提醒注意：模板的声明或定义只能在全局，命名空间或类范围内进行。即不能在局部范围，函数内进行，比如不能在**main**函数中声明或定义一个模板。

**三、模板的形参**

**有三种类型的模板形参：类型形参，非类型形参和模板形参。**

**1、类型形参**

　　1.1 、类型模板形参：**类型形参由关见字class或typename后接说明符构成**，如**template<class T> void h(T a){}**;其中**T**就是一个类型形参，类型形参的名字由用户自已确定。**模板形参表示的是一个未知的类型**。模板类型形参可作为类型说明符用在模板中的任何地方，与内置类型说明符或类类型说明符的使用方式完全相同，即可以用于指定返回类型，变量声明等。

**作者原版：1.2、** 不能为同一个模板类型形参指定两种不同的类型，比如**template<class T>void h(T a, T b){}，**语句调用**h(2, 3.2)**将出错，因为该语句给同一模板形参**T**指定了两种类型，第一个实参**2**把模板形参T指定为**int**，而第二个实参**3.2**把模板形参指定为**double**，两种类型的形参不一致，会出错。**（针对函数模板）**

　　作者原版：1.2针对函数模板是正确的，但是忽略了类模板。下面将对类模板的情况进行补充。

**本人添加1.2补充版（针对于类模板）、**当我们声明类对象为：**A<int> a**，比如**template<class T>T g(T a, T b){}**，语句调用**a.g(2, 3.2)**在编译时不会出错，但会有警告，因为在声明类对象的时候已经将**T**转换为**int**类型，而第二个实参**3.2**把模板形参指定为**double**，在运行时，会对**3.2**进行强制类型转换为**3**。当我们声明类的对象为：**A<double> a**,此时就不会有上述的警告，因为从**int**到**double**是自动类型转换。

**演示示例１：**

**TemplateDemo.h**

[复制代码](javascript:void(0);)

1 #ifndef TEMPLATE\_DEMO\_HXX

2 #define TEMPLATE\_DEMO\_HXX

3

4 template<class T> class A{

5 public:

6 T g(T a,T b);

7 A();

8 };

9

10 #endif

[复制代码](javascript:void(0);)

**TemplateDemo.cpp**

[复制代码](javascript:void(0);)

1 #include<iostream.h>

2 #include "TemplateDemo.h"

3

4 template<class T> A<T>::A(){}

5

6 template<class T> T A<T>::g(T a,T b){

7 return a+b;

8 }

9

10 void main(){

11 **A<int> a;**

12 **cout<<a.g(2,3.2)<<endl;**

13 }

[复制代码](javascript:void(0);)

**编译结果：**

[复制代码](javascript:void(0);)

1 --------------------Configuration: TemplateDemo - Win32 Debug--------------------

2 Compiling...

3 TemplateDemo.cpp

4 G:\C++\CDaima\TemplateDemo\TemplateDemo.cpp(12) : **warning C4244: 'argument' : conversion from 'const double' to 'int', possible loss of data**

5

6 TemplateDemo.obj - **0 error(s), 1 warning(s)**

[复制代码](javascript:void(0);)

**运行结果：　　　　　　5**

　　我们从上面的测试示例中可以看出，并非作者原作中的那么严密！此处仅是本人跟人测试结果！请大家本着实事求是的态度，自行验证！

**2、非类型形参**

2.1 、非类型模板形参：**模板的非类型形参也就是内置类型形参**，如**template<class T, int a> class B{};**其中**int a**就是非类型的模板形参。

2.2、 非类型形参在模板定义的内部是常量值，也就是说非类型形参在模板的内部是常量。

**2.3、 非类型模板的形参只能是整型，指针和引用，像double，String, String \*\*这样的类型是不允许的。**但是**double &，double \*，**对象的引用或指针是正确的。

2.4、 **调用非类型模板形参的实参必须是一个常量表达式**，即他必须能在编译时计算出结果。

2.5 、注意：**任何局部对象，局部变量，局部对象的地址，局部变量的地址都不是一个常量表达式，都不能用作非类型模板形参的实参**。全局指针类型，全局变量，全局对象也不是一个常量表达式，不能用作非类型模板形参的实参。

2.6、 **全局变量的地址或引用，全局对象的地址或引用const类型变量是常量表达式，可以用作非类型模板形参的实参**。

2.7 、**sizeof**表达式的结果是一个常量表达式，也能用作非类型模板形参的实参。

2.8 、当模板的形参是整型时调用该模板时的实参必须是整型的，且在编译期间是常量，比如**template <class T, int a> class A{};**如果有**int b，这时A<int, b> m;**将出错，因为**b**不是常量，如果**const int b，这时A<int, b> m**;就是正确的，因为这时**b**是常量。

**2.9 、非类型形参一般不应用于函数模板中**，比如有函数模板**template<class T, int a> void h(T b){}**，若使用**h(2)**调用会出现无法为非类型形参a推演出参数的错误，对这种模板函数可以用显示模板实参来解决，如用h<int, 3>(2)这样就把非类型形参a设置为整数3。显示模板实参在后面介绍。

2.10、 非类型模板形参的形参和实参间所允许的转换  
**1、允许从数组到指针，从函数到指针的转换。如：template <int \*a> class A{}; int b[1]; A<b> m;即数组到指针的转换**  
**2、const修饰符的转换。如：template<const int \*a> class A{}; int b; A<&b> m;   即从int \*到const int \*的转换。**  
**3、提升转换。如：template<int a> class A{}; const short b=2; A<b> m; 即从short到int 的提升转换**  
**4、整值转换。如：template<unsigned int a> class A{};   A<3> m; 即从int 到unsigned int的转换。**  
**5、常规转换。**

**非类型形参演示示例1：**

**由用户自己亲自指定栈的大小，并实现栈的相关操作。**

**TemplateDemo.h**

[复制代码](javascript:void(0);)

1 #ifndef TEMPLATE\_DEMO\_HXX

2 #define TEMPLATE\_DEMO\_HXX

3

4 template<class T,int MAXSIZE> class Stack{//MAXSIZE由用户创建对象时自行设置

5 private:

6 T elems[MAXSIZE]; // 包含元素的数组

7 int numElems; // 元素的当前总个数

8 public:

9 Stack(); //构造函数

10 void push(T const&); //压入元素

11 void pop(); //弹出元素

12 T top() const; //返回栈顶元素

13 bool empty() const{ // 返回栈是否为空

14 return numElems == 0;

15 }

16 bool full() const{ // 返回栈是否已满

17 return numElems == MAXSIZE;

18 }

19 };

20

21 template <class T,int MAXSIZE>

22 Stack<T,MAXSIZE>::Stack():numElems(0){ // 初始时栈不含元素

23 // 不做任何事情

24 }

25

26 template <class T,int MAXSIZE>

27 void Stack<T, MAXSIZE>::push(T const& elem){

28 if(numElems == MAXSIZE){

29 throw std::out\_of\_range("Stack<>::push(): stack is full");

30 }

31 elems[numElems] = elem; // 附加元素

32 ++numElems; // 增加元素的个数

33 }

34

35 template<class T,int MAXSIZE>

36 void Stack<T,MAXSIZE>::pop(){

37 if (numElems <= 0) {

38 throw std::out\_of\_range("Stack<>::pop(): empty stack");

39 }

40 --numElems; // 减少元素的个数

41 }

42

43 template <class T,int MAXSIZE>

44 T Stack<T,MAXSIZE>::top()const{

45 if (numElems <= 0) {

46 throw std::out\_of\_range("Stack<>::top(): empty stack");

47 }

48 return elems[numElems-1]; // 返回最后一个元素

49 }

50

51 #endif

[复制代码](javascript:void(0);)

**TemplateDemo.cpp**

[复制代码](javascript:void(0);)

1 #include<iostream.h>

2 #include <iostream>

3 #include <string>

4 #include <cstdlib>

5 #include "TemplateDemo.h"

6

7 int main(){

8 try {

9 Stack<int,20> int20Stack; // 可以存储20个int元素的栈

10 Stack<int,40> int40Stack; // 可以存储40个int元素的栈

11 Stack<std::string,40> stringStack; // 可存储40个string元素的栈

12

13 // 使用可存储20个int元素的栈

14 int20Stack.push(7);

15 std::cout << int20Stack.top() << std::endl; //7

16 int20Stack.pop();

17

18 // 使用可存储40个string的栈

19 stringStack.push("hello");

20 std::cout << stringStack.top() << std::endl; //hello

21 stringStack.pop();

22 stringStack.pop(); //Exception: Stack<>::pop<>: empty stack

23 return 0;

24 }

25 catch (std::exception const& ex) {

26 std::cerr << "Exception: " << ex.what() << std::endl;

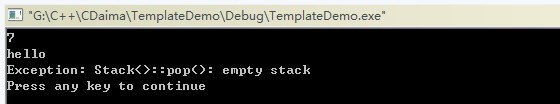
27 return EXIT\_FAILURE; // 退出程序且有ERROR标记

28 }

29 }

[复制代码](javascript:void(0);)

**运行结果：**



**非类型形参演示示例2：**

**TemplateDemo01.h**

[复制代码](javascript:void(0);)

1 #ifndef TEMPLATE\_DEMO\_O1

2 #define TEMPLATE\_DEMO\_01

3

4 template<typename T> class CompareDemo{

5 public:

6 int compare(const T&, const T&);

7 };

8

9 template<typename T>

10 int CompareDemo<T>::compare(const T& a,const T& b){

11 if((a-b)>0)

12 return 1;

13 else if((a-b)<0)

14 return -1;

15 else

16 return 0;

17 }

18

19 #endif

[复制代码](javascript:void(0);)

**TemplateDemo01.cpp**

[复制代码](javascript:void(0);)

1 #include<iostream.h>

2 #include "TemplateDemo01.h"

3

4 void main(){

5 CompareDemo<int> cd;

6 cout<<cd.compare(2,3)<<endl;

7 }

[复制代码](javascript:void(0);)

**运行结果：　　　　　-1**

[复制代码](javascript:void(0);)

1 #include<iostream.h>

2 #include "TemplateDemo01.h"

3

4 void main(){

5 CompareDemo<double> cd;

6 cout<<cd.compare(3.2,3.1)<<endl;

7 }

[复制代码](javascript:void(0);)

**运行结果：　　　　　　1**

**TemplateDemo01.h** **改动如下：**

[复制代码](javascript:void(0);)

1 #ifndef TEMPLATE\_DEMO\_O1

2 #define TEMPLATE\_DEMO\_01

3

4 template<typename T> class CompareDemo{

5 public:

6 int compare(T&, T&);

7 };

8

9 template<typename T>

10 int CompareDemo<T>::compare(T& a,T& b){

11 if((a-b)>0)

12 return 1;

13 else if((a-b)<0)

14 return -1;

15 else

16 return 0;

17 }

18

19 #endif

[复制代码](javascript:void(0);)

**TempalteDemo01.cpp**

按 Ctrl+C 复制代码



按 Ctrl+C 复制代码

**非类型形参演示示例3：**

**TemplateDemo02.cpp**

[复制代码](javascript:void(0);)

1 #include<iostream.h>

2

3 template<typename T>

4 const T& max(const T& a,const T& b){

5 return a>b ? a:b;

6 }

7

8 void main(){

9 cout<<max(2.1,2.2)<<endl;//模板实参被隐式推演成double

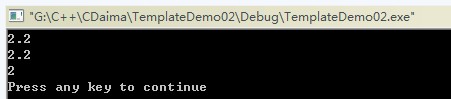
10 cout<<max<double>(2.1,2.2)<<endl;//显示指定模板参数。

11 cout<<max<int>(2.1,2.2)<<endl;//显示指定的模板参数，会将函数函数直接转换为int。

12 }

[复制代码](javascript:void(0);)

**运行结果：**



**cout<<max<int>(2.1,2.2)<<endl;**//显示指定的模板参数，会将函数函数直接转换为int。此语句会出现警告：

[**子类构造、析构时调用父类的构造、析构函数顺序**](http://www.cnblogs.com/liushui-sky/p/5798667.html)

析构函数调用的次序是先派生类的析构后基类的析构，也就是说在基类的的析构调用的时候,派生类的信息已经全部销毁了。

而**定义一个对象时先调用基类的构造函数、然后调用派生类的构造函数；析构的时候恰好相反：先调用派生类的析构函数、然后调用基类的析构函数。**

原因：

1 派生类构造函数中的某些初始化可能是基于基类的，所以规定构造在类层次的**最根处**开始，而在**每一层，首先调用基类构造函数，然后调用成员（此处的成员只指各种类对象如QString a，不含基本类型变量如int n、指针变量如QString \*a）对象构造函数**（因为C++的成员变量是不会自动初始化的，只能使用初始化列表初始化（调用成员的构造函数，如果不在初始化列表中显式调用的话，则会隐式调用成员变量的默认构造函数，通过汇编可以发现）或在本层构造函数内初始化）        参考：http://www.cnblogs.com/lidabo/p/3790606.html）。

2 如果没有显式调用基类的构造函数，会自动调用基类的无参构造函数。而如果基类只有带参数的构造函数，则会报错。不一定要显式的无参构造函数，可以显式调用基类带参数的构造函数。

[复制代码](javascript:void(0);)

#include<iostream>

using namespace std;

class Base{

public:

Base(int c){cout<< "基类带参构造函数" << c << endl;}

~Base(){cout<<"基类析构" << endl;}

};

class Derived:public Base{

public:

Derived(int c):Base(c) // 显式调用基类构造函数

{

cout<< "派生类带参构造函数" << c << endl;

}

~Derived(){cout<<"派生类析构" << endl;}

};

int main()

{

int i = 9;

Derived d1(i);

return 0;

}

[复制代码](javascript:void(0);)

输出结果：

基类带参构造函数9

派生类带参构造函数9

派生类析构

基类析构