**1.虚函数与纯虚函数：**

定义一个函数为**虚函数**，不代表函数为不被实现的函数。定义他为虚函数是为了允许用基类的指针来调用子类的这个函数。

定义一个函数为**纯虚函数**，才代表函数**没有被实现**。定义**纯虚函数**是为了实现一个接口，起到一个**规范**的作用，规范**继承**这个类的程序员必须实现这个函数。

假设我们有下面的类层次：

class A

{

public:

virtual void foo()

{

cout<<"A::foo() is called"<<endl;

}

};

class B:public A

{

public:

void foo()

{

cout<<"B::foo() is called"<<endl;

}

};

int main(void)

{

A \*a = new B();

a->foo(); // 在这里，a虽然是指向A的指针，但是被调用的函数(foo)却是B的!

return 0;

}

这个例子是**虚函数**的一个典型应用，虚函数只能借助于指针或者引用来达到多态的效果。

**C++纯虚函数定义**

纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加 =0:

virtual void func1() const = 0;

（1）=0说明它是纯虚函数

（2）const表明该函数不能修改其数据成员

**2.include防范与#pragma once**

在C和C++编程语言中，#include防范，有时被称作宏防范，用于处理#include指令时，可避免重复引入的问题。

以下的C语言程序展示了缺少#include防范时会出现的问题：

**文件“grandfather.h”**

struct foo {

int member;

};

**文件“father.h”**

#include "grandfather.h"

**文件“child.c”**

#include "grandfather.h"

#include "father.h"

此处child.c间接引入了两份grandfather.h头文件中的内容。明显可以看出，foo结构被定义两次，因此会造成编译错误。

**使用#include防范**

**文件“grandfather.h”**

#ifndef GRANDPARENT\_H

#define GRANDPARENT\_H

struct foo {

int member;

};

#endif

**文件“father.h”**

#include "grandfather.h"

**文件“child.c”**

#include "grandfather.h"

#include "father.h"

此处grandfather.h第一次被引入时会定义宏GRANDPARENT\_H。当father.h再次引入grandfather.h时，#ifndef测试失败，编译器会直接跳到#endif的部分，也避免了第二次定义foo结构。程序也就能够正常编译。

**困难**

为了让#include防范正确运作，**每个防范都必须检验并且有条件地设置不同的前置处理宏**。因此，使用了#include防范的方案必须**制订一致性的命名方法**，并确定这个方法不会和其他的头文件或任何可见的全局变量冲突。

为了解决这个问题，许多C和C++程序开发工具提供非标准的指令**#pragma once**。#pragma once是一个非标准但是被广泛支持的前置处理符号，会让所在的文件在一个单独的编译中只被包含一次。以此方式，#pragma once提供类似include防范的目的，但是拥有较少的代码且能避免名称的碰撞。

**使用示例：**

**grandparent.h**

#pragma once

struct foo

{

int member;

};

**parent.h**

#include "grandparent.h"

**child.c**

#include "grandparent.h"

#include "parent.h"

使用#pragma once代替include防范将加快编译速度，因为这是一种高端的机制；编译器会自动比对文件名称或inode而不需要在头文件去判断#ifndef和#endif。另一方面，部分编译器，例如GCC、clang等，也包含特别的代码来识别和有效率的管理include防范。因此使用#pragma once并不会得到明显的加速。

此外，因为编译器自己承担管理#pragma once，它不必定义新的指令名称，例如在include防范中示例的GRANDFATHER\_H。这能排除名称碰撞的风险。

然而，这种高端的管理有好也有坏；设计者必须依赖编译器正确的管理#pragma once。编译器如果犯错，例如没有辨认出在相同文件中的两个不同符号链接名称指针，此时编译会错误。编译器对于#pragma once可能包含相关的bug等。

**3.override 方法重写**

在面向对象的编程中，方法重写是一种语言功能，它允许子类提供其超类或父类之一已经提供的方法的特定实现。它允许特定类型的多态性。通过提供与父类中的方法具有相同名称，相同参数或签名以及相同返回类型的方法，子类中的实现将覆盖超类中的实现。执行的方法的版本将由调用该方法的对象确定。

4.**子类RowMatrix使用父类Matrix的成员变量需加上Matrix::**

C++标准中规定（14.6.2 3），一个非受限的名称查找的时候将不会考虑依赖型的基类。所以

一个模板子类其实是不能在实例化(运行时)之前就知道他的模板父类到底是谁，**即子类看不到父类成员**。查找不到(编译时)就会错误。解决办法是把它变成一个依赖型名称：

在父类变量x前加父类名称：Parent::x;