**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;

**5**.**throw Exception(ExceptionType::OUT\_OF\_RANGE,"index is out of range");**中Exception以及ExceptionType定义在src\include\common\exception.h中。

**6.NULL、0、nullptr的区别**

**C的NULL**

在C语言中，我们使用NULL表示空指针，也就是我们可以写如下代码：

int \*i = NULL;

foo\_t \*f = NULL;

实际上在C语言中，NULL通常被定义为如下：

#define NULL ((void \*)0)

也就是说NULL实际上是一个void \*的指针，然后吧void \*指针赋值给int \*和foo\_t \*的指针的时候，**隐式转换**成相应的类型。而如果换做一个C++编译器来编译的话是要出错的，因为C++是**强类型**的，void \*是不能隐式转换成其他指针类型的，所以通常情况下，编译器提供的头文件会这样定义NULL：

#ifdef \_\_cplusplus ---简称：cpp c++ 文件

#define NULL 0

#else

#define NULL ((void \*)0)

#endif

**C++的0**

因为C++中不能将void \*类型的指针隐式转换成其他指针类型，而又为了解决空指针的问题，所以C++中**引入0来表示空指针**（注：0表示还是有缺陷、不完美），这样就有了类似上面的代码来定义NULL。实际上C++的书都会推荐说C++中更习惯使用0来表示空指针而不是NULL，尽管NULL在C++编译器下就是0。为什么C++的书都推荐使用0而不是NULL来表示空指针呢？我们看一个例子：

在foo.h文件中声明了一个函数：

void bar(sometype1 a, sometype2 \*b);

这个函数在a.cpp、b.cpp中调用了，分别是：

**a.cpp:**

bar(a, b);

**b.cpp:**

bar(a, 0);

好的，这些代码都是正常完美的编译运行。但是突然在某个时候我们功能扩展，需要对bar函数进行扩展，我们使用了**重载**，现在foo.h的声明如下：

void bar(sometype1 a, sometype2 \*b);

void bar(sometype1 a, int i);

这个时候危险了，a.cpp和b.cpp中的调用代码这个时候就不能按照期望的运行了。但是我们很快就会发现b.cpp中的**0是整数**，也就是在**overload resolution(重载解析)**的时候，我们知道它**调用的是void bar(sometype1 a, int i)这个重载函数**，于是我们可以做出如下修改让代码按照期望运行：

bar(a, static\_cast<sometype2 \*>(0));

**这时调用的就是void bar(sometype1 a,sometype2 \*b)这个重载函数。**如果我们一开始就有bar的这两个**重载函数**的话，我们会在一开始就想办法避免这个问题（不使用重载）或者我们写出正确的调用代码，然而后面的这个重载函数或许是我们几个月或者很长一段时间后加上的，那我们出错的可能性就会加大了不少。貌似我们现在说道的这些跟C++通常使用0来表示空指针没什么关系，好吧，假设我们的调用代码是这样的：

**foo.h**

void bar(sometype1 a, sometype2 \*b);

void bar(sometype1 a, int i);

**a.cpp**

bar(a, b);

**b.cpp**

bar(a, NULL);

当bar的重载函数在后面加上来了之后，我们会发现**bar(a, NULL);**调用了**void bar(sometype1 a, int i);**但是出错的时候，我们找到b.cpp中的调用代码也很快可能**忽略**过去了，因为我们用的是NULL空指针啊，应该是调用的**void bar(sometype1 a, sometype2 \*b)**这个重载函数啊。但实际上NULL在C++中就是0，写NULL这个反而会让你没那么**警觉**，因为NULL不够“明显”，而这里如果是使用**0来表示空指针**，那就会够“明显”，因为0是空指针，但它更是一个整形常量。

**总结：在C++中，使用0做为空指针会比使用NULL做空指针让你更加警觉。**

**C++ 11的nullptr**

虽然上面我们说明了0比NULL可以让我们更加警觉，但是我们并没有避免这个问题。这个时候C++ 11的**nullptr**就很好的解决了这个问题，我们在C++ 11中使用**nullptr**来表示空指针，这样最早的代码是这样的，

**foo.h**

void bar(sometype1 a, sometype2 \*b);

**a.cpp**

bar(a, b);

**b.cpp**

bar(a, nullptr);

在我们后来把bar的重载加上了之后，代码是这样：

**foo.h**

void bar(sometype1 a, sometype2 \*b);

void bar(sometype1 a, int i);

**a.cpp**

bar(a, b);

**b.cpp**

bar(a, nullptr);

这时候，我们的代码能够如预期的一样正确运行。

**7.static关键字**

C++的static有两种用法：**面向过程**程序设计中的static和**面向对象**程序设计中的static。前者应用于普通变量和函数，不涉及类；后者主要说明static在类中的作用。

**1.面向过程设计中的static**

**1.1静态全局变量**

在**全局变量**前，加上关键字static，该变量就被定义成为一个**静态全局变量**。我们先举一个静态全局变量的例子，如下：

//Example 1

#include <iostream.h>

void fn();

static int n; //定义静态全局变量

void main()

{

　　n=20;

　　cout<<n<<endl;

　　fn();

}

void fn()

{

　　 n++;

　　cout<<n<<endl;

}

静态全局变量有以下特点：

• 该变量在**全局数据区**分配内存；

• **未经初始化的静态全局变量**会被程序自动初始化为0（**自动变量**的值是随机的，除非它被显式初始化）；

• 静态全局变量在声明它的整个文件都是可见的，而在文件之外是不可见的；

静态变量都在**全局数据区**分配内存，包括后面将要提到的**静态局部变量**。对于一个完整的程序，在内存中的分布情况如下：

**代码区**

**全局数据区**

**堆区**

**栈区**

一般程序的由new产生的动态数据存放在**堆区**，函数内部的自动变量存放在**栈区**。自动变量一般会随着函数的退出而**释放空间**，静态数据（即使是函数内部的**静态局部变量**）也存放在全局数据区。全局数据区的数据并不会因为函数的退出而释放空间。细心的读者可能会发现，Example 1中的代码中将 “static int n; //定义静态全局变量”改为“int n; //定义全局变量”。程序**照样正常运行**。的确，定义全局变量就可以实现变量在文件中的共享，但定义静态全局变量还有以下好处：

• 静态全局变量**不能被其它文件所用**；

• 其它文件中**可以定义相同名字的变量，不会发生冲突**；

您可以将上述示例代码改为如下：

//Example 2

//File1

#include <iostream.h>

void fn();

static int n; //定义静态全局变量

void main()

{

　　n=20;

　　cout<<n<<endl;

　　fn();

}

//File2

#include <iostream.h>

extern int n;

void fn()

{

　　 n++;

　　cout<<n<<endl;

}

编译并运行Example 2，您就会发现上述代码可以分别通过编译，但运行时出现错误。试着将 “static int n; //定义静态全局变量”改为 “int n; //定义全局变量”

再次编译运行程序，细心体会“全局变量”和"静态全局变量"的区别。

**1.2.静态局部变量**

在局部变量前，加上关键字static，该变量就被定义成为一个静态局部变量。 我们先举一个静态局部变量的例子，如下：

//Example 3

#include <iostream.h>

void fn();

void main()

{

　　fn();

　　fn();

　　fn();

}

void fn()

{

　　static n=10;

　　cout<<n<<endl;

　　 n++;

}

通常，在函数体内定义了一个变量，每当程序运行到该语句时都会给该局部变量分配栈内存。但随着程序退出函数体，系统就会收回栈内存，局部变量也相应失效。但有时候我们需要在两次调用之间对变量的值进行保存。通常的想法是定义一个全局变量来实现。但这样一来，变量已经不再属于函数本身了，不再仅受函数的控制，给程序的维护带来不便。

**静态局部变量**正好可以解决这个问题。静态局部变量保存在**全局数据区**，而不是保存在栈中，每次的值保持到下一次调用，直到下次**赋新值**。

静态局部变量有以下特点：

• 该变量在**全局数据区**分配内存；

• 静态局部变量在程序执行到该对象的声明处时被**首次初始化**，以后的函数调用不再进行初始化；

• 静态局部变量一般在声明处初始化，如果**没有显式初始化**，会被程序自动初始化为0；

• 它始终驻留在**全局数据区**，直到程序运行结束。但其作用域为**局部作用域**，当定义它的函数或语句块结束时，其作用域随之结束；

**1.3静态函数**

在函数的返回类型前加上static关键字,函数即被定义为静态函数。静态函数与普通函数不同，它只能在声明它的文件当中可见，**不能被其它文件使用**。

静态函数的例子：

//Example 4

#include <iostream.h>

static void fn();//声明静态函数

void main()

{

　　fn();

}

void fn()//定义静态函数

{

　　int n=10;

　　cout<<n<<endl;

}

定义静态函数的好处：

• 静态函数不能被其它文件所用；

• 其它文件中可以定义相同名字的函数，不会发生冲突；

**二、面向对象的static关键字（类中的static关键字）**

2.1静态数据成员

在类内数据成员的声明前加上关键字static，该数据成员就是类内的静态数据成员。先举一个静态数据成员的例子。

//Example 5

#include <iostream.h>

class Myclass

{

public:

　　Myclass(int a,int b,int c);

　　void GetSum();

private:

　　int a,b,c;

　　static int Sum;//声明静态数据成员

};

int Myclass::Sum=0;//定义并初始化静态数据成员

Myclass::Myclass(int a,int b,int c)

{

　　this->a=a;

　　this->b=b;

　　this->c=c;

　　Sum+=a+b+c;

}

void Myclass::GetSum()

{

　　cout<<"Sum="<<Sum<<endl;

}

void main()

{

　　Myclass M(1,2,3);

　　M.GetSum();

　　 Myclass N(4,5,6);

　　N.GetSum();

　　M.GetSum();

}

可以看出，静态数据成员有以下特点：

• 对于非静态数据成员，**每个类对象**都有自己的拷贝。而静态数据成员被当作是**类的成员**。无论这个类的对象被**定义了多少个**，静态数据成员在程序中也**只有一份拷贝**，由该类型的所有对象共享访问。也就是说，静态数据成员是该类的所有对象所**共有**的。对该类的多个对象来说，静态数据成员只分配一次内存，供所有对象共用。所以，静态数据成员的值对每个对象都是一样的，它的值可以更新；

• 静态数据成员存储在**全局数据区**。静态数据成员定义时要**分配空间**，所以**不能在类声明中定义**。在Example 5中，语句int Myclass::Sum=0;是定义静态数据成员；

• 静态数据成员和普通数据成员一样遵从public,protected,private访问规则；

• 因为静态数据成员在全局数据区分配内存，属于本类的所有对象共享，所以，**它不属于特定的类对象**，在没有产生类对象时其作用域就可见，即在没有产生类的实例时，我们就可以操作它；

• 静态数据成员初始化与一般数据成员**初始化不同**。静态数据成员初始化的格式为：

＜数据类型＞＜类名＞::＜静态数据成员名＞=＜值＞

• 类的静态数据成员有**两种访问形式**：

＜类对象名＞.＜静态数据成员名＞ 或 ＜类类型名＞::＜静态数据成员名＞

如果静态数据成员的访问权限允许的话（即public的成员），可在程序中，按上述格式来引用静态数据成员；

• 静态数据成员主要用在**各个对象都有相同的某项属性**的时候。比如对于一个存款类，每个实例的利息都是相同的。所以，应该把利息设为存款类的静态数据成员。这有两个好处，第一，不管定义多少个存款类对象，利息数据成员都共享分配在全局数据区的内存，所以节省存储空间。第二，一旦利息需要改变时，只要改变一次，则所有存款类对象的利息全改变过来了；

• 同**全局变量**相比，使用静态数据成员有两个优势：

1. 静态数据成员没有进入程序的全局名字空间，因此不存在与程序中其它全局名字**冲突**的可能性；

2. 可以实现信息隐藏。静态数据成员可以是private**成员**，而全局变量不能；

**2.2静态成员函数**

与静态数据成员一样，我们也可以创建一个静态成员函数，它为**类的全部**服务而不是为某一个类的**具体对象**服务。静态成员函数与静态数据成员一样，都是类的内部实现，属于类定义的一部分。普通的成员函数一般都隐含了一个this指针，this指针指向类的对象本身，因为普通成员函数总是具体的属于某个类的具体对象的。通常情况下，this是缺省的。如函数fn()**实际上是this->fn()**。但是与普通函数相比，静态成员函数由于不是与任何的对象相联系，因此它**不具有this指针**。从这个意义上讲，它**无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数**，它**只能调用其余的静态成员函数**。下面举个静态成员函数的例子。

//Example 6

#include <iostream.h>

class Myclass

{

public:

　　Myclass(int a,int b,int c);

　　static void GetSum();/声明静态成员函数

private:

　　int a,b,c;

　　 static int Sum;//声明静态数据成员

};

int Myclass::Sum=0;//定义并初始化静态数据成员

Myclass::Myclass(int a,int b,int c)

{

　　this->a=a;

　　this->b=b;

　　this->c=c;

　　Sum+=a+b+c; //非静态成员函数可以访问静态数据成员

}

void Myclass::GetSum() //静态成员函数的实现

{

　　// cout<<a<<endl; //错误代码，a是非静态数据成员

　　cout<<"Sum="<<Sum<<endl;

}

void main()

{

　　Myclass M(1,2,3);

　　M.GetSum();

　　Myclass N(4,5,6);

　　N.GetSum();

　　Myclass::GetSum();

}

关于静态成员函数，可以总结为以下几点：

• 出现在类体外的函数定义不能指定关键字static；

• 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＞）

调用类的静态成员函数。

**8.unique\_ptr**

unique\_ptr是c ++ 11提供的智能指针实现之一，用于**防止内存泄漏**。unique\_ptr对象包含一个原始指针，并负责其生命周期。当这个对象被销毁的时候，它的析构函数会删除关联的原始指针。unique\_ptr有重载的- >和\*运算符，所以它可以像普通的指针那样被使用。

get()方法：获取当前 unique\_ptr 指针内部包含的普通指针。

**9.auto关键字**

auto是C++程序设计语言的关键字。自C++11以来，auto关键字用于两种情况：声明变量时根据初始化表达式自动推断该变量的类型、声明函数时函数返回值的占位符。C++98标准中auto关键字用于自动变量的声明，但由于使用极少且多余，在C++11中已删除这一用法。

auto可以在声明变量时根据变量初始值的类型自动为此变量选择匹配的类型。

举例：对于值x=1；即可以声明：int x = 1或long x = 1，也可以直接声明auto x = 1。

在类型冗长复杂、变量使用范围专一时，可以使用auto关键字使得程序更清晰易读。

**10.本地测试**

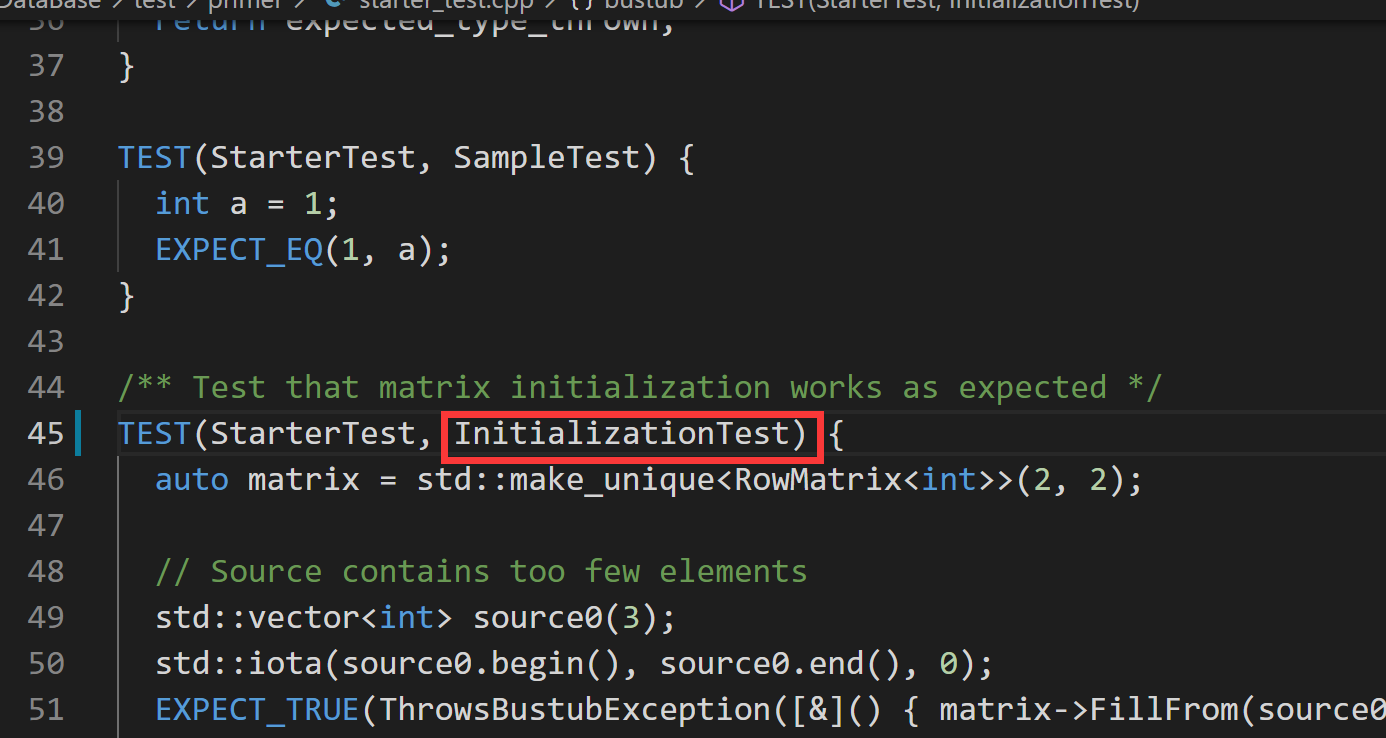
在test/primer/starter\_test.cpp文件中，在下图所示处原来有DISABLED，去掉。然后执行以下命令：

$ cd build

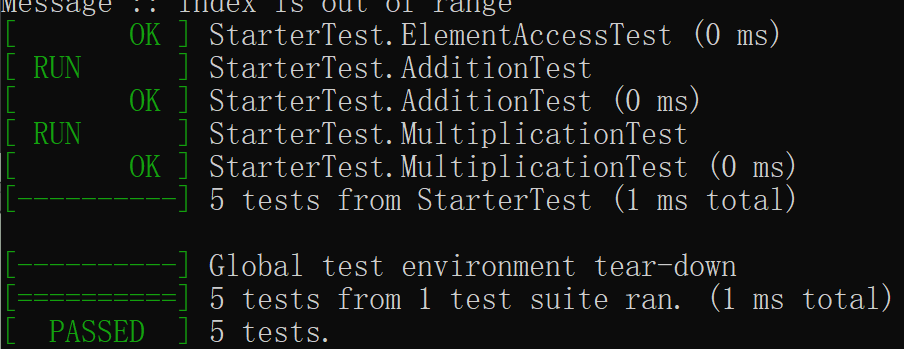
$ make starter\_test

$ ./test/starter\_test

即可完成本地测试。



本地测试通过则如下图：



11.代码格式

运行下面命令，使得代码符合Google编程规范

$ make format

$ make check-lint

$ make check-clang-tidy