**C++学习经验**

顾名思义，本学习经验的总结基于Essential\_C++书中的知识，同时包括自己的理解。

Reference语义

1. 首先我不想使这部分的总结和对指针的理解混为一谈，但是我觉得它确实与指针有几分相似。
2. 先列举我们常见的定义变量的方法：

int a = 1024;

int \*pa = &a;

int &ra = a;

前两种并不陌生，第三种便是我们今天的主角，就这样就表示声明了一个reference。

1. **这种定义方式可以实现怎么个事？**

它主要是可以让你**像使用a一样去使用ra**。

int &ra = a;写出来之后ra的值就是a的值，就是1024。

之后如果我们这样写：

ra = 2048;

那么ra的值就是2048，与此同时a的值也会是2048，就是这么个事儿。

初始接触这个概念会有些许不解，时间长了就好。

简单解释就是我们可以把ra理解为a的一个副本、影子等（怎么好理解怎么来），改变了ra就会相应的改变a，**像是ra与a建立了链接一样，而且这种链接是可以运用到函数中的。**

**为什么说可以运用到函数中呢？**

这就是它与指针相似的地方，假设我们写一个交换值的函数swap()

void swap(int &x , int &y)

{

int temp = x;

x = y;

y = temp;

}

可见参数列表中运用了我们的新知识，这样写函数，我们只需要给swap函数传入两个变量即可，而不是传入两个变量的地址。**重要的是这样写可以实现像使用指针那样交换函数外的两个变量的值，即和用指针写的函数效果相同。**

1. reference的声明必须被初始化，因为我们无法在除初始化的其它地方将一个变量与它建立链接。

也就是说不可以int &ra; ra = a;这样写，必须要给它一个变量从而进行初始化，也就是说需要这样写int &ra = a;我们可能会想为什么必须要这样，答案很简单，因为我们知道当我们写过int &ra = a;之后我们对ra的操作都会对a产生相同的影响，哪怕是赋值语句也都是在执行给a赋值，因为ra和a已经建立了链接，它不能通过赋值的操作建立某一个变量与ra的链接。也就是说int &ra; ra = a;这种写法根本不能表示ra与a建立链接的意思，最多表示把a的值赋给ra但是ra又和谁建立了链接，这个段代码并没有表明，所以这是不正确的写法。

**为函数提供默认参数值**

1、C++允许我们为函数中全部或部分的参数设置默认值，也就是初值。

2、初值的作用就是如果我们调用了这个函数，但是没有在被设置初值的参数的位置传入某个参数，那么函数工作时就会使用你设置的初值。

3、**为函数提供默认参数值有两个规则：**

（1）**默认值的解析的解析操作从参数列表的最右边开始**，也就是说一旦你开始为某一个参数设定默认值，那么**参数列表中位于这个参数右边的参数也全部需要设定默认值。**

错误写法：void display(ostream &os = cout , const vector<int> &vec)

位于参数os被设置了默认值为cout那么右边的vec就也需要设置默认值，但是上述做法并不是我们期望的那样。

正确写法：void display(const vector<int> &vec , ostream &os = cout)

1. 默认值只能够指定一次，可以写在函数定义的地方，也可以写在函数声明的地方。

（补充一点：**函数定义**的地方通常放在**程序代码文件**中，而**函数声明**的地方通常在某个**头文件**中，个人认为写在函数声明的地方更容易我们对其进行修改或是检查，因此我个人现阶段支持写在函数声明中）

**重载函数**

1. 重载函数的功能与后面要将到的function template（模板函数）有相似之处，但并不完全相同，也没有谁大谁小之分。
2. 重载函数，帮助我们实现了使用**同一个函数**名称解决**不同的参数列表**，说得再多不如看代码。
3. 假设我们要写一个display函数，让它帮我们输出参数列表中的数据到屏幕上，我们可能会这样写：

void display(int& x)

{

cout << x << endl;

}

这样写的话，如果我们要输出string类型的字符串，我们就得再写一个函数实现这个功能还要保证函数名不重复，或者如果我们要一次输出两个整型参数，我们得又写一个输出两个参数的函数，函数名称还不能与原有函数名称相同，这不仅效率很低，还不利于我们对函数的使用，因为我们需要记忆大量的函数名字，并且清晰的辨认出每个名字对应的功能。

重载函数就可以解决这类问题，C++允许我们使用同一个函数名称写不同的函数：

void display(int& x)

{

cout << x << endl;

}

void display(string& x)

{

cout << x << endl;

}

void display(int& x , int& y)

{

cout << x << endl;

cout << y << endl;

}

如上所示，我们用了同一个函数名称display，写了三个函数，它们具有不同的功能，当我们尽管随意调用函数，函数再编译链接时可以识别到你具体要用哪个功能。

1. **关于重载函数有几点需要注意的地方：**
2. 编译器无法根据函数的返回类型来区分两个具有相同名称的函数。意思就是

void display(int& x)和int display(int& x)，编译器是无法区分的，**区分具有相同名称的函数的唯一方式就是看它们的参数列表是否相同，参数列表相同，那么就是同一个函数**，所以不要写多个参数列表重复的函数，这是不规范的，甚至是错误的行为。

1. **重载函数的返回类型不一定非要相同**，我们也可以这样写：

void display(int& x)

int display(int& x , int& y)

bool display(string& x)

也就是说**重载函数不对函数的返回类型有什么限制或要求**，只要合理即可。

**模板函数的定义和使用**

1. function template（模板函数），学习完这方面的知识之后，我们便可以学会很好的将其与重载函数的知识联系起来，写出更加强大、更具**兼容性**的函数。
2. 首先我先举例说明一下function template可以解决什么问题，在**重载函数**部分，我们可能写过这样的函数：

void display(int& x)

{

cout << x << endl;

}

void display(string& x)

{

cout << x << endl;

}

void display(double& x)

{

cout << x << endl;

}

像这样的函数，它们除了参数列表中的数据类型不同，其余地方完全一样，像这样的三个函数，我们完全没有必要为三个数据类型去写三个函数，否则那么多数据类型，程序员绝对要忙死了。function template就可以解决这个问题。

1. function template的定义，是这样写的：

template <typename elemType>

先写一个关键字template，空格之后使用尖括号<>里面**通常**（为什么说“通常”以后就会理解）包含两部分，typename也是关键字，空格之后我们可以随意写一个名字，秉持着程序员的好习惯，我们写elemType（代表“元素类型”，其中elem指element：元素）。

**目前个人水平已知，我们必须为每一个我们想使用模板函数写法的函数写一次上述定义。**

因此，我们需要这样写：

template <typename elemType>

void display(elemType& x)

{

cout << x << endl;

}

也就是在函数头的前面紧跟着就要写function template的定义，注意函数头中参数列表的变化。

这样写出来的函数就可以适用于我们常用的int、double、float、string等基础类型，即我们**调用函数时不需要在意传入的参数是什么类型**，只用放心使用即可，**函数会自己识别传入的参数时什么类型**，**我们的elemType就表示那个我们希望推迟决定的（或者说暂时还不确定的）数据类型**（当然，它也可以是其他名字，不一定非要elemType，只是这样可读性强一些）。

1. **原理：**

假设我们调用函数：

int main()

{

int a = 0;

display(a);

}

编译器会将elemType绑定为int类型（因为a是int类型的），然后产生一份display()函数实例，于是其参数的类型就变成int，其他数据类型同理。

1. 一般而言，函数具有多种实现方式，我们可以将它重载（overload），其每份实例提供的是相同的通用服务。如果我们希望让程序代码的主体不变，仅仅改变其中用到的数据类型，可以通过function template达到目的。

**初步了解Iterator（泛型指针）**

1、Iterator（泛型指针）又称作迭代器，基于我个人目前对它的理解，**迭代器可以帮助我们更加灵活的去操控一个容器中的元素，就像C语言中对数组使用指针那样。**

2、**介绍这部分内容之前我想先讲一下本节的背景：**

这部分背景其实就是Essential\_C++书中3.1节的大致内容，具体就是我们想要写一个函数find()，其功能是在容器中找到我们想要的某个数据，参数列表有三个参数，分别是**容器的头**、**容器的尾**、**我们想要寻找的数据**，不难想到用**指针**、**reference**以及**function template**这三方面的知识可以很好的解决问题，于是我们写出了这样的函数：

template <typename elemType>

elemType \*find(elemType \*first , elemType \*last , elemType &value)

{

if(!first || !last)

{

cout << "wrong" << endl;

return 0;

}

for( ; first != last ; first++)

{

if( \*first == value)

{

return first;

}

}

return 0 ;

}

之后，考虑到在传入vector容器时vector有可能为空的情况，又写了两个函数对vector容器为空的问题进行了解决，代码如下：

template <typename elemType>

inline elemType \*begin( vector<elemType> &vec )

{

return vec.empty() ? 0 : &vec[0];

}

template <typename elemType>

inline elemType \*end( vector<elemType> &vec )

{

return vec.empty() ? 0 : &vec[vec.size()];

}

Vector容器乃至其他许多容器都带有像empty()、size()、front()、back()等函数，便捷我们对容器的使用。

上述两个自定义函数分别为begin和end主要功能就是判断vector是否为空，如果是空的话empty()会返回true反之则为false；如果不空，begin函数会返回vector容器的第一个元素的地址，end函数则会返回**最后一个元素的下一个地址**，因为这样可以方便的使用迭代器，具体原因相信看完本节会有答案。

3、ok，背景已经基本阐述完毕，注意到find函数里是通过for循环以及一些运算符（++、=、<、>等）来实现容器中元素的更迭，这如果用于连续的储存单元里的数据完全没有问题，但是我们知道list这种容器其本质可以看作双向链表，链表的储存单元并不连续，它们只是通过指针被连在一起，所以我们无法对一个list容器的元素使用++或其他运算符从而使其更迭为该元素的下一个元素，一般情况下我们做不到，迭代器就是为了帮助我们做到这件事。

4、**迭代器如何定义？**

试想一下，定义迭代器，我们需要告诉迭代器什么信息，大概需要两个重要的要素：

1. 迭代对象（某个容器）的类型，这可以让迭代器知道应该以什么方式去访问下一元素。
2. 上述容器中元素的类型，这可以让迭代器进行提领操作时返回正确的值（是int、string、还是float或double等等）

因此我们定义迭代器时会这样写：

vector<int>::iterator iter;

不难看出其格式就是：

**容器类型<元素（数据）类型>::itertor 名称**

那么对于上述语句，iter就是我们自定义的一个迭代器名字，这个迭代器是元素类型为int的vector容器的迭代器。

双冒号（：：）表示此迭代器（iterator）是位于int vector定义内的嵌套类型，以后会讲到更本质的原理，在第四章部分，会实现自己的iterator class，目前仅仅停留在应用层面。

5、C++中每一个标准容器，都提供有一个名为begin的操作函数，可返回一个iterator，指向第一个元素；另一个为名为end操作函数，返回一个iterator，指向最后一个元素的下一个位置。

我们可以对iterator对象进行赋值、递增递减、比较、提领等操作，其中如果iterator对象指向某一个元素，那么我们对其进行提领操作之后它就表示该元素的值，具体是什么值，

因容器的元素类型而异。

至此，我想我们大概知道应该如何让list容器的元素仅仅通过++就可以更迭到下一个元素了吧。没错，就是这样写：

list<int> ilist;

list<int>::iterator it = ilist.begin();

假设我们定义了一个list容器，数据类型为int的对象ilist。

那么第二行代码就是定义了一个相应类型的iterator对象，名为it，这个it此时指向ilist的第一个元素，\*it则表示第一个元素的具体的值。

如果元素类型是string的话，我们还可以**使用it调用底部的string元素所提供的操作**，只需使用“箭头”（arrow）即可：it->size()。

见下一页

6、最终我们这个find函数就变成了这个样子：

template <typename IteratorType , typename elemType>

IteratorType find(IteratorType first , IteratorType last , elemType &value)

{

for( ; first != last ; first++)

{

if( \*first == value)

{

return first;

}

}

return last;

}

我们首先可以从这段代码中认识到，function template其实也是可以这样在先括号里写多个typename的。

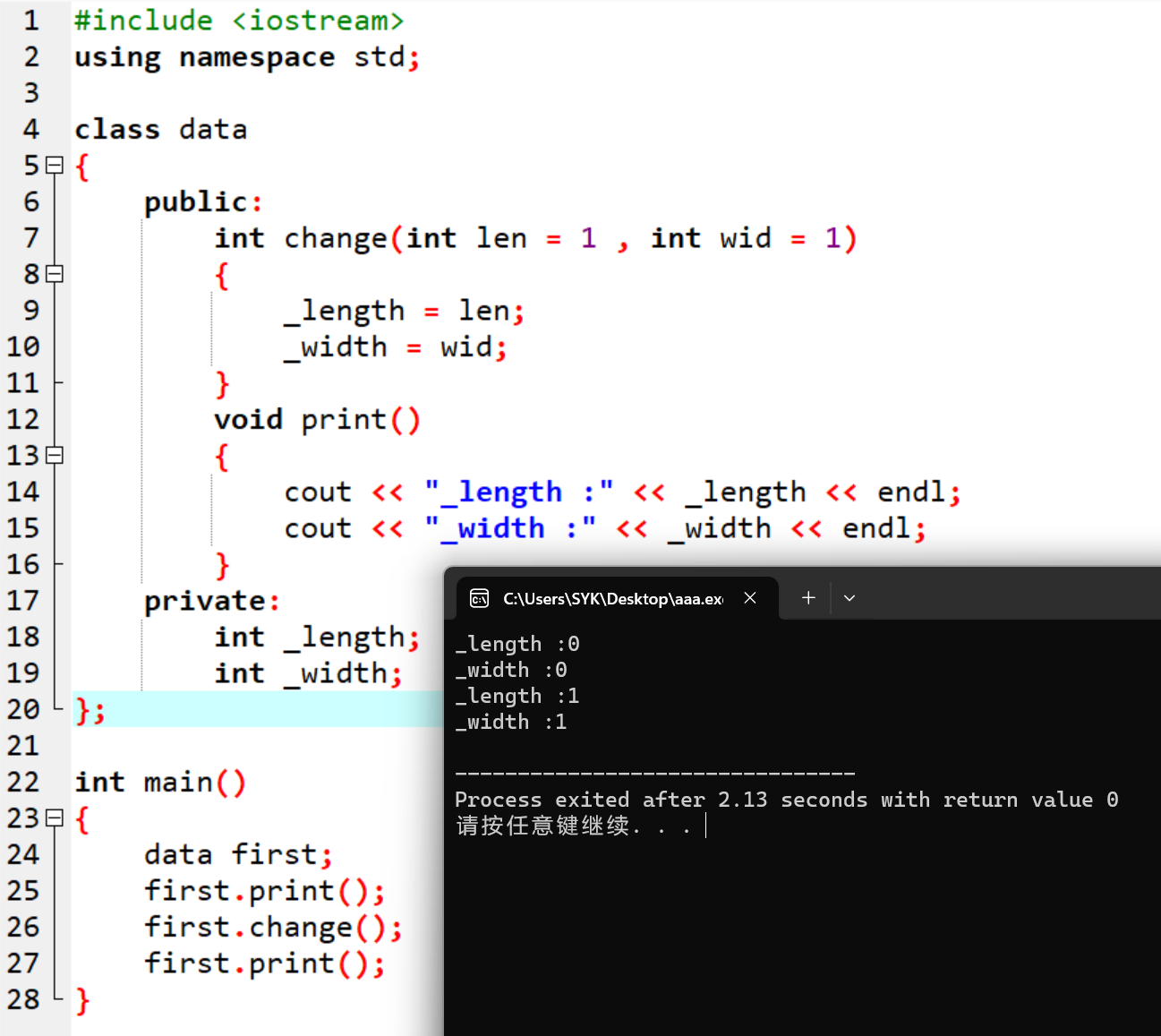
这里面的IteratorType和elemType仅仅是为了阅读方便而设置的名字，只是一个名字而已，并没有实际的功能，不要看代码把自己看晕了，我们当然也可以写其它名字。

写了两个typename的主要目的就是为了区分不同的类型，我们需要把iterator对象和需要找的数据传入find函数，这就已经涉及到了两种不同的类型一种是iterator而另一种是基本的数据类型，同时也需要函数返回iterator对象，因此我们需要两个typename来进行区分，只是让编译器知道哪些地方是不同的即可，**函数头写的是IteratorType但是它只是一个名字，我们实际使用时传入什么类型的迭代器对象、传入什么类型的数据，IteratorType就是什么类型，elemType就是什么类型**。

现阶段我们可以把它看作是容器的“指针”，像使用指针一样使用它。注意，只是“看作”，迭代器本质未必和我们熟知的指针是一回事。

**mutable（可变）和const（不变）**

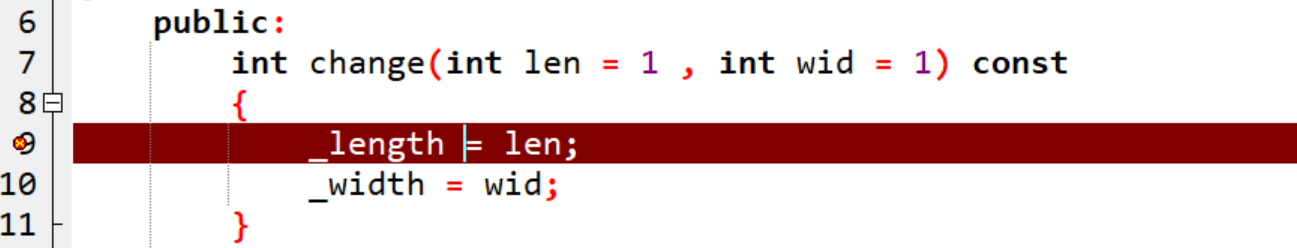
1. 在此所讲的mutable和const与学习C语言时认识到的const不同，我们这里的可变与不变是针对的**一个类（class）中的变量（可以是public也可以是private）是否可以被其成员函数（member function）所改变**。因此主要讨论member function中const的用法。
2. 过多的文字解释只会使文章晦涩难懂，直接看例子：



上图展示的是在没有const加入的时候我们的成员函数change正常工作的情况，它给我们的两个数据变量\_length 和\_width进行了赋值。也就是说我们允许change对class object也就是对first内部的数据的值进行改变。

但是在某些情况下我们可能只想使用，不想因为某个函数的调用而改变了class object中的数据值。这时就需要使用const对member function做一些手脚。

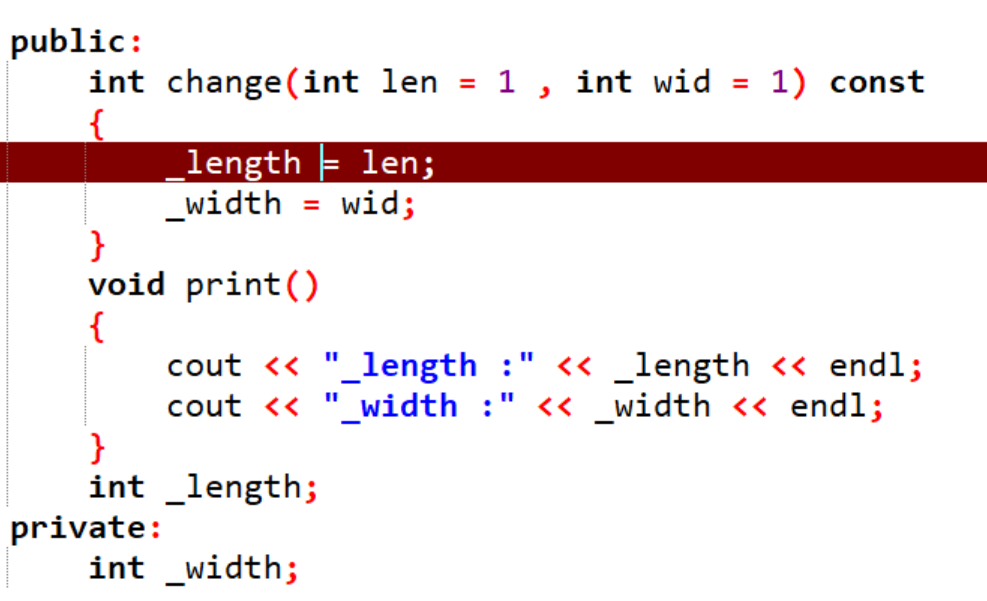
1. 请看下图：





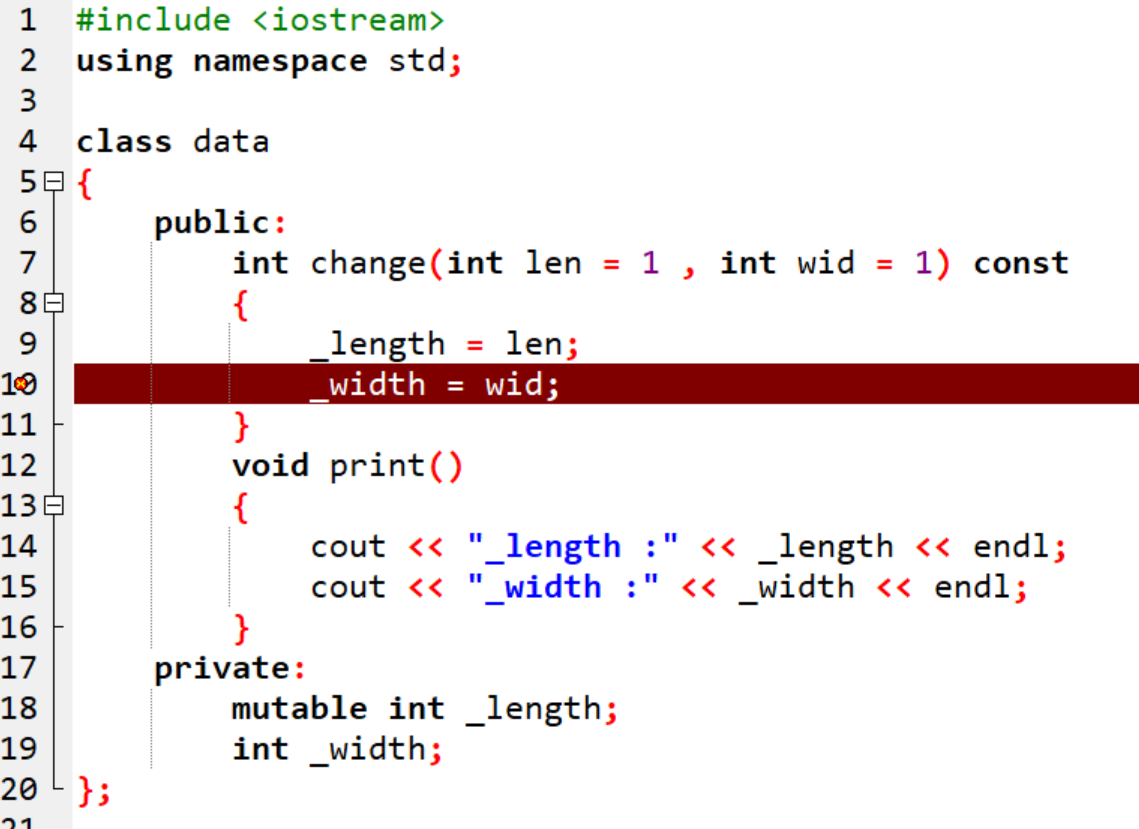
其他代码不变，只在change函数的参数列表后面写上一个const，编译器就会去检查我们的change函数是否真的存在某些可能改变我们的变量的值的句子，很明显，编译器报了两个错误都是在说**我们在只读对象中为成员赋值**。

也就是说加了const之后我们的这个函数就不应该存在可能改变data这个类的数据的值的语句，具体到代码的话就是说我们的change函数不可以对\_length 和\_width进行赋值了。当然这不局限于private中的变量，public中的变量也一视同仁。只是本例由于习惯并没有写在public中，见下图，写在public中确实是也会出错的。

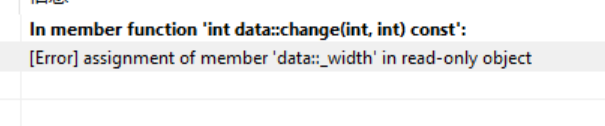


4、好了接下来说一下mutable的作用，其实先讲const就是为了mutable做铺垫，更便于我们理解mutable的功能。

**其实我认为mutable就仿佛是给人一种在const的基础上为某些变量开放权限的感觉。**



可以看到上图中大部分代码没有做出改变，我们写的const也依旧存在，只是不一样的是出现了一句：mutable int \_length;这时我们发现编译器报错的位置与前文那一张报错的图片报错的位置有所不同，这次并没有对\_length的赋值语句进行报错，而是只有一个\_width的报错。



可以看出上图对于报错的具体的说明中也只有\_width的报错，没有了曾经出现过的\_length的报错。这就是mutable的作用。

**mutable可以为我们想要改变其只读性质的变量开放权限，使其可以被改变、被赋值。**这是我个人的理解，但是我觉得应该没有什么问题。

**构造函数（constructor）和析构函数（destructor）**

埋坑！！！

**静态类成员与静态成员函数**

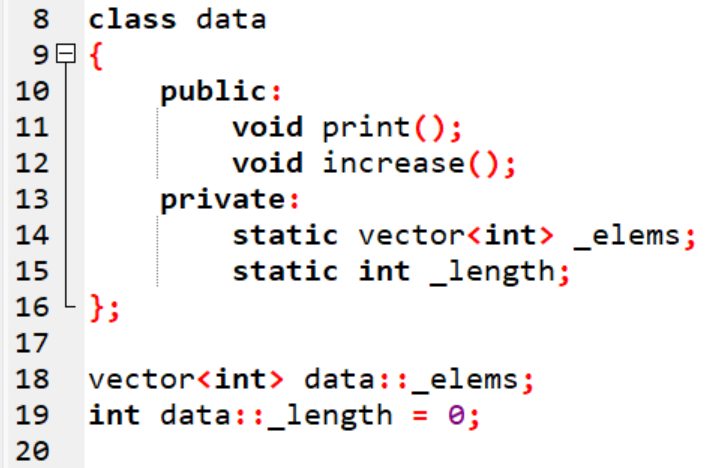
**前言：**

在讲本节之前我想先种下一个核心的贯穿整节的要点，那就是不论静态类成员还是静态成员函数它们其实都是属于class，而不是属于class创造的对象。

具体的说就是，通常情况下我们的变量写在class里，那么class创造出来的每一个对象它们内部的同名的变量是没有联系的，你的是你的我的是我的。但是静态类成员就不一样，它隶属于class不属于class创造出来的对象，所以所有的这个class的对象共享这一个静态类成员，任何一个对象都可以改变它的数据。静态成员函数就看下面的讲解吧。

**静态类成员：**

1、静态类成员只需要我们对class中的普通的成员变量用static修饰即可，但是这不是完整的静态类成员的设置，请看下图：



在class中我们写了两个静态类成员，但是仅仅写到第16行还不足以运行代码，因为这样写出来的仅仅只是它们的声明，还缺少**必要的**定义也就是18、19两行代码。我们需要**在class（类）外**写出静态类成员的定义，也就是初始化。其实这比较像我们在class内写member function的声明，然后在class外写member function的定义，与这个比较相似。

2、**有关静态类成员的一些要点：**

（1）首先就是，**静态类成员不可以在类内部进行初始化，也不可以在类的构造函数中进行初始化**，因为类的构造函数是为每一个对象服务的，想想前言的核心观念，静态类成员属于class而不是对象，所以不可能创造一个对象，构造函数就初始化一次静态类成员，这样不就乱套了。

（2）静态类成员不可以用初始化列表方式进行初始化（至于什么是初始化列表方式，请参考“**构造函数与析构函数**”一节）

（3）**静态成员函数在类外实现时只需在类内声明时加static，类外实体无须加static关键字，否则是错误的**（思考下为什么？）。因为如果在类外添加static关键字，编译器会将其当作第二种理解，也就是全局静态函数。

（4）静态类成员如果不初始化，则默认为0，不初始化不是说18，19行代码不需要写了，而是说类似19行那样的“= 0”不写的话，默认值为0。

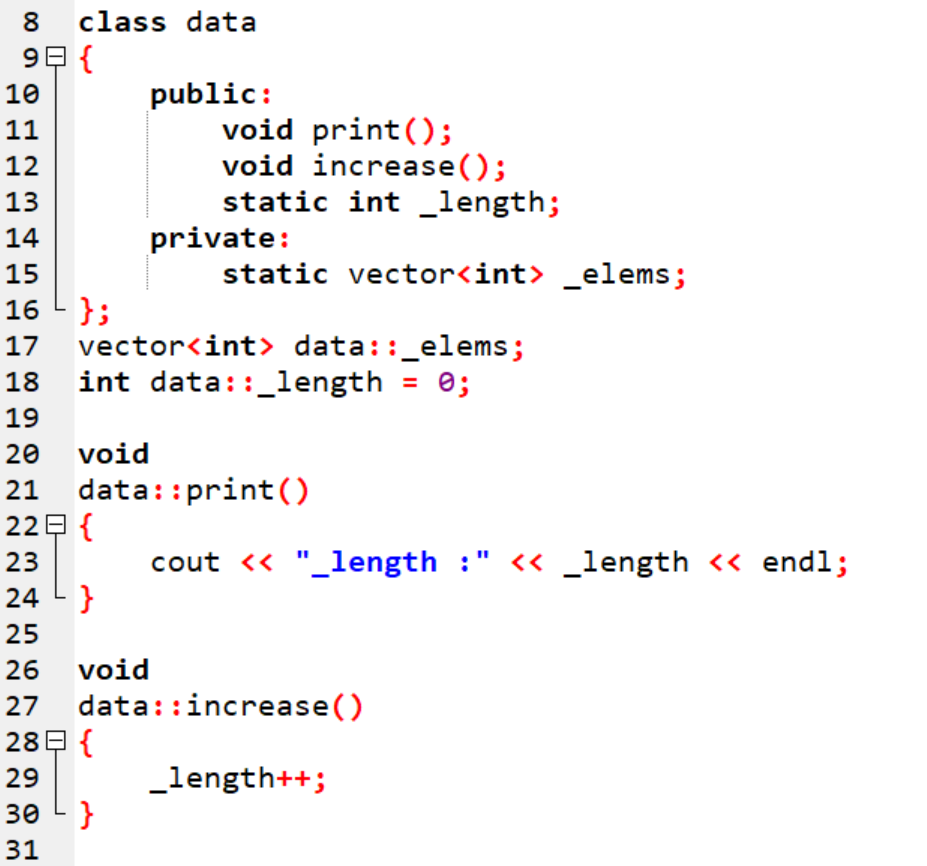
（5）静态成员仍然遵循public，private，protected访问准则。

（6）静态成员变量在编译链接时分配，程序加载时被落实到内存中，程序结束时死亡，等同于全局变量

（7）静态成员变量在对象中不占用存储空间

（8）静态成员可以直接通过类名调用（注意调用时，成员是public还是private）

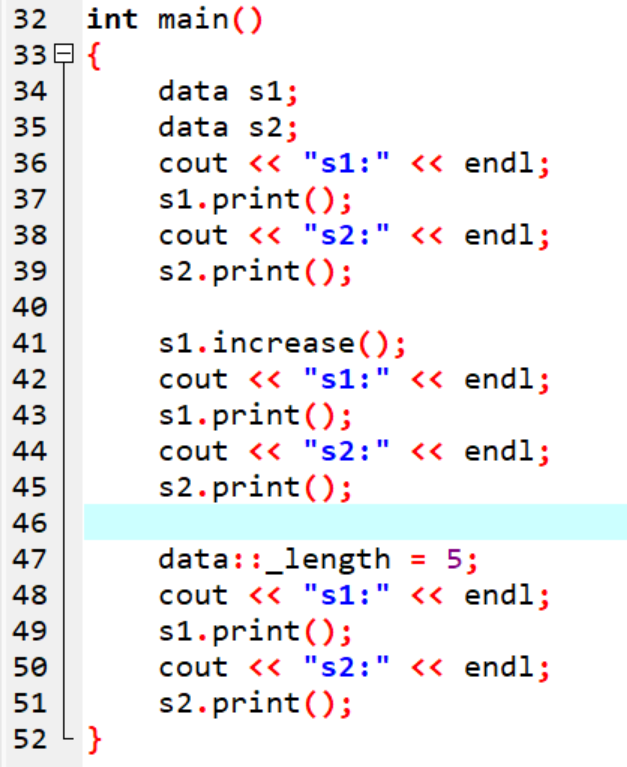
3、接下来看一个例子结束对静态类成员的讲解吧：



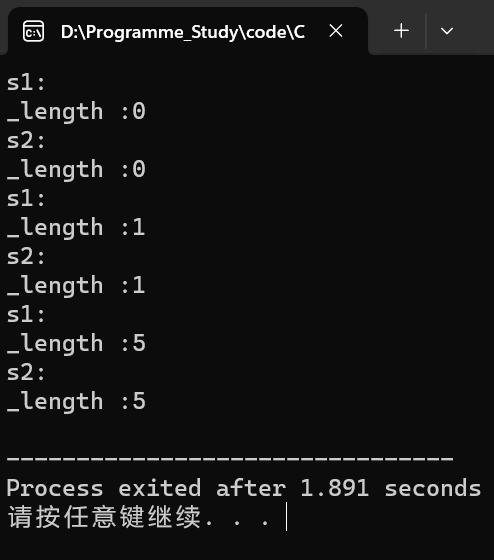
这是data类的定义以及静态类成员的声明和定义，可以看到有两个函数，print函数就是打印\_length的数值，increase函数用于增长\_length的数值。

下面是main函数：

见下一页



main函数创建了两个对象s1和s2首先先打印两个对象的\_length不出意外，都是0。之后我们只使用的s1的increase函数去增长\_length，但是打印的结果是两个对象的\_length数值都为1，这就印证了我们的核心观念里面的静态类成员属于class而非class对象，所有对象共享静态类成员。在接下来就是测试一下使用类名直接调用静态类成员，可以想象打印的结果二者的数值都是5。下面看一下输出结果：



**静态成员函数：略**

运算符重载

1. 重载运算符函数的基本格式：

二元：成员函数：（即在类中定义时）

返回类型 operator Op(类类型右操作数)

二元：非成员函数：（即在类外定义时）

返回类型 operator Op(类类型左操作数 , 类类型右操作数)

（其中“类类型”是指重载运算符的类）

一元：成员函数：

类类型& operator Op( )

一元：非成员函数：

类类型& operator Op(类类型& 对象)

2、如果要在类中重载所有的比较运算符，除了我们一个一个去定义即自己写之外，还可以借助C++编译器自带的头文件<utility>，这个头文件为运算符函数<=、>、>=和!=定义了模板，来比较类型T的两个对象。该模板根据小于和等于运算符来定义这些运算符，也就是说，使用这个头文件的话，我们只需要在类中定义小于和等于运算符就可以了，其他运算符的重载在需要使用它们时，编译器会使用模板生成所需的运算符，优点就是这些模板函数在不使用它们的程序中不会生成，所以不会占用内存。

当然，如果类中有自己定义的其他运算符的重载，那么编译器会调用你自己写的，而不会调用该运算符的模板。

3、我们知道，当类中有指针类型的数据成员的时候，类的拷贝构造函数（又称副本构造函数）的实现是必要的。它可以避免浅拷贝，实现深拷贝，也就是在复制指针成员的同时，也会复制一份与被复制对象大小一样的内存空间，这样就能保证不会内存泄漏。

但是我们需要思考的是只有拷贝构造函数需要实现深拷贝吗？

其实赋值运算符也会出现浅拷贝的问题，这不难理解，但是容易忘记，我们将一个类的对象赋值给另一个对象时，如果是以浅拷贝的方式赋值，那可能就会出现同一个地方被析构两次这种事情，这是我们不希望看到的。

因此，如果类有指针类型的数据成员，那么我们既需要实现副本构造函数，也需要实现赋值运算符的重载，使它们具有深拷贝的作用。

在实现赋值运算符重载时有些需要注意的知识点：

基本格式：

类名& operator=(const 类名& 自定义参数名)

需要注意的是，返回类型和参数列表都需要使用引用，这样可以避免不必要的复制，因为将变量或对象按值传入函数时，会复制一个同样的副本，但是这个复制是浅拷贝，因此需要避免这类问题。

另一个需要注意的点就是，可能出现用一个对象去赋值它自己的情况，形如：box = box，这种情况会使原对象被“漂白”，导致其内存不能释放，换言之就是内存泄露。

解决方法就是在函数体的开头就检查左右操作数是否相同，如果不相同就执行正常的赋值操作，如果相同，就不执行任何操作，直接返回“\*this”

3、重载下标运算符是一个比较重要且难懂的主题，在此写一段话作为标记，由于进度需要加快，所以暂时不进行这方面的深造。

**函数对象**

1、其实这部分内容在书上是属于运算符重载的，但是我个人认为需要单独开一篇笔记，它还是很重要的。

2、