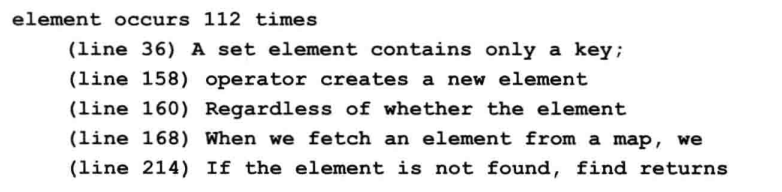
# 学习文章

### 文本查询器(Text query)

目标：查询文本中有没有这个单词，并且返回单词所在的行，和那些行的内容



比如查询 element这个单词 它出现了112次 并且把那些行都显示出来了；

### 《C++ Primer》

### 关联容器

**有8种 其实就是map 和set**

map 普通map

mutimap 就是可重复的map

unordered\_map 就是无序的map

unordered\_mutimap 就是无序的且可重复的map

然后set同理，所以一共有8个

可以使用列表初始化

// justForTest.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。

#include <C:\Users\DELL\source\repos\LibrarySystem\dataStructure.cpp>

#include <vector>

#include <string>

#include <iostream>

#include <map>

#include <set>

#include <unordered\_map>

#include <unordered\_set>

using namespace std;

int main()

{

set<int> testset = { 1,2,3 };

map<string, int> mp;

//初始化 set和map都是模板类型的

//map的元素是pair对

mp = { {"annotation",1} ,{"double",2} };

//map的下标操作，map的下标操作比较特殊，如果是这样子的，annotation关键字对应的值（键值对）本来是1，就会赋2

mp["annotation"] = 2;

mp["trible"] = 3; //这样会先查找，找不到啊，那会插入trible，然后赋值0，然后再赋值3

/\*那map的查找操作，应该用find 或者count 那么find很好，如果没找到 就会返回尾迭代器\*/

mp.find("annotation");

//所以就有了find==end 这种好操作 要是等于end，就意味着没找到

if(mp.find("annotation1 ")==mp.end())

cout << "没找到" << endl;

//打印map的每一个键值对

for (const auto& x : mp)

{

cout << "关键字是" << x.first << " 值是" << x.second << endl;

}

testset.insert(3);

testset.insert(5);

for (auto it=testset.begin();it!=testset.end();it++)

cout << \*it;

//迭代器 cbegin() 和begin() 这个cbegin是const

/\* begin() Return iterator to beginning (public member function )

cbegin() Return const\_iterator to beginning (public member function )

\*/

/\*

set的迭代器是const的 只能用迭代器来访问这些只读元素 不能用迭代器来修改

\*/

//set插入元素用insert可以是两个迭代器，也可以直接是；

/\* key\_type 是键值对的键的类型 value\_type是键值对的值的类型

且对于set来说键，值都是一样的类型

所以set的这两个都相同

但是对于map来说，就不一样了

map的key那个好理解，就是键的类型

但是map的value\_type却是指的是pair这种键值对的类型

那么值的类型，我们用了一个特别的mapped\_tyepe

\*/

map<string, int>::mapped\_type v1 = 2;

map<string, int>::value\_type v2 = { "string",1 };

// map<string, int>::value\_type v2 = 1; 这句就会报错

map<string, int>::key\_type v3 = " stign";

}

### 类之间共享数据，类的生存周期的问题



问题：比如类A要用类B的数据，但是如果类B的对象在A之前就销毁了，那么A就没办法拿到类B的数据了

就好比，在handler所属的类MAP\_GR\_DIS 的对象销毁了

这个C++ Primer里有说用share\_ptr 来解决

Share\_ptr 就像inode这种东西 ，要是世界上所以人都不记得你了，你就真的被析构，删除了

### 图书管理系统

1.遇到的问题，实现查询功能的时候 从文件里查询记录，这个时候文件是按行存储的，这时候就要做类似解析器一样的事情；

//可以用kmp算法来匹配buffer里有没有bookName

2. // justForTest.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。

//这是用来测试string方法里面的find函数的

Find函数的标号 string 从0开始计数 第二个参数 如果是5 则从第5个数字开始检索

#include <string>

#include <iostream>

using namespace std;

int main()

{

string buffer;

buffer = "hello#2#3#";

//0123456789

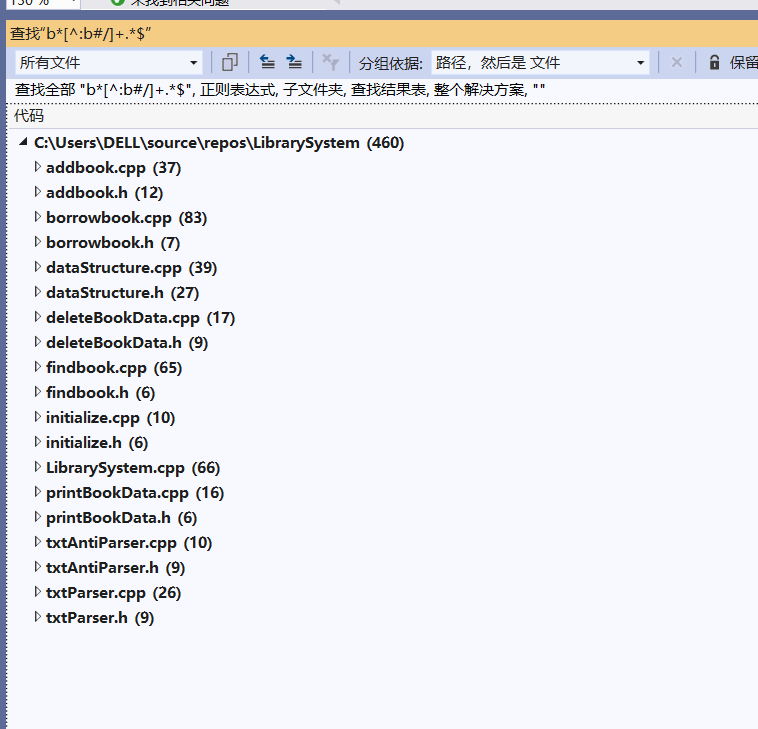
cout<<buffer.find('#',8);

//test for find

}

3. 查找所有代码行数，包括系统生成  
1.打开项目。  
2.编辑-查找和替换-选中“在文件中查找”  
然后就会弹出一个查找框，输入正则表达式  
b\*[^:b#/]+.\*$  
选择查找整个解决方案，查找选项选择使用正则表达式，然后查找全部，最后就会统计出一共有几行。

我这个一共460行



### 虚函数

#### 对虚函数的理解

虚函数：基类指针在指向不同派生类对象时的进行对this指针的强制类型转换调用不同虚函数的实现版本

虚函数：作用于类体系的动态联编依赖基类指针指向派生类对象，调用虚函数的不同版本

基类指针 基类对象 子类指针 子类对象

这四个之间有4种关系嘛

我们主要研究

基类指针 指向不同对象 的情况

基类指针指向不同对象时 都是会调用基类的那个函数

如果我们想要基类指针随着指向的对象的改变而调用不同的函数

就要在基类里把那个函数声明为虚函数

当然 在派生类中 那些函数也默认为虚函数

其中例子我觉得上面讲到的都有体现的就是C++程序设计基础书中P323 例9-26

还有一件事 你知道为什么构造函数不能是虚函数 ，而析构函数可以是虚函数吗？

其实这一个知识点与前面的知识是互相印证 ，还能助于我们对前面知识的理解。

构造函数 必须沿着构造函数链层层调用 都得调用 而不是按着 虚函数的特性 选择性的调用 就比如一个最经典的例子 Point2d 在Point2d的构造函数里 调用了Point的构造函数

Point3d在point3d的构造函数里调用了 Point2d的构造函数 而在调用Point2d的构造函数时又调用了Point的构造函数。

而析构函数 呢 ，其实只需要根据对象的类型 合理的选择正确的析构函数就可了啊 这不正是虚函数的特性吗？

#### 其他

（1）什么是多态性（注意：多态与多态性是两种概念）

多态性是指具有不同功能的函数可以使用相同的函数名，这样就可以用一个函数名调用不同内容的函数。在面向对象方法中一般是这样表述多态性：向不同的对象发送同一条消息，不同的对象在接收时会产生不同的行为（即方法）。总而言之就是在不考虑对象的类型的情况下直接使用对象。

（1）增加了程序的灵活性  
　　以不变应万变，不论对象千变万化，使用者都是同一种形式去调用，如func(animal)  
（2）增加了程序额可扩展性  
　　通过继承animal类创建了一个新的类，使用者无需更改自己的代码，还是用func(animal)去调用

### 宏定义

#include <iostream>

#define Max(a,b,c) \

(a?c:d);

int main()

{

int a = 1;

int c = 2;

int d = 100;

Max(a, c, d)

}

注意，这里 main函数里面max后面没接分号 因为在宏定义里已经有了；

#include <iostream>

#define Max(a,b,c) \

(a?c:d)

int main()

{

int a = 1;

int c = 2;

int d = 100;

Max(a, c, d)

std::cout << "Hello World!\n";

}

但是这样就是不行的 因为宏定义那里没有

### 输出运算符<<的重载

#include <iostream>

using namespace std;

class pointer

{

public:

pointer(float x)

{

\_x = x;

}

public:

float x() { return \_x; }

private:

float \_x;

public:

void x(float val)

{

\_x = val;

}

};

inline ostream& operator<<(ostream& os, pointer & pt)

{

os << "(" << pt.x() << ")" ;

return os;

};

int main()

{

pointer m\_instance(10);

double a;

cout << m\_instance;

}

基本形式是固定的 ，就是这个样子

ostream& operator<<(ostream& os, pointer & pt)

三个都是引用 参数是输出流ostream的引用 和 对象的引用

### 三层class的结构 并且运用了上面的知识

#include <iostream>

using namespace std;

class Point

{

public:

Point(float fpx):\_x(fpx)

{

}

private :

float \_x;

public:

float x() {

return \_x;

}

};

class Point2d :public Point

{

public :

Point2d(float fpx,float fpy) : Point(fpx),\_y(fpy) {};

private:

float \_y;

public:

auto y()

{

return \_y;

}

};

class Point3d :public Point2d

{

private :

float \_z;

public:

float z()

{

return \_z;

}

public :

Point3d(float fpx,float fpy,float fpz):Point2d(fpx,fpy),\_z(fpz){}

};

void operator<<(ostream& os, Point3d& obj)

{

os << obj.x()<< endl << obj.y() << endl << obj.z() << endl;

}

int main()

{

Point3d test(10, 20, 30);

cout << test;

}

其中特地测试了一下重载运算符 返回类型改成void 可以成功运行

### 运用了类模板技术

#include <iostream>

template <typename type>

class Point3d

{

public :

type \_x;

type \_y;

type \_z;

public:

Point3d(type fpx, type fpy, type fpz)

{

\_x = fpx;

\_y = fpy;

\_z = fpz;

}

};

using namespace std;

void operator<<(ostream& os, Point3d<int> & obj)

{

os << obj.\_x << endl << obj.\_y << endl << obj.\_z << endl;

}

int main()

{

Point3d<int> test(10, 20, 30);

cout << test;

}

无聊试了一下类模板

### C++对象模式

类的数据成员有两种 ： static \ nonstatic

类的函数有两种 : virtual 、 static 、nonstatic