第七讲 标准模版库

一、C++标准新特性

二、字符串类

三、标准模版库(STL)

四、迭代器

五、算法

六、STL示例代码

一、C++标准新特性

1.1 命名空间namespace

namespace即命名空间或名字空间，所谓namespace，是指标识符的各种可见范围。C++标准程序库中的所有标识符都被定义于一个名为std的namespace中。它的目的就是为了解决同一作用域下同名变量或函数的冲突，比如两个人写的库文件中出现同名的变量或函数(不可避免)，使用起来就有问题了。为了解决这个问题，引入了名字空间这个概念，通过使用 namespace xxx；你所使用的库函数或变量就是在该名字空间中定义的，这样一来就不会引起不必要的冲突了。

命名空间的成员必须使用命名空间和操作符的名称 ::

命名空间的定义使用关键字namespace。

定义一个命名空间:

```
namespace my_space{
    int count;
    const int size = 10;
    const double pi = 3.14;
    void print( );

    //命名空间的定义是可以嵌套的
    namespace inner{
        int count; //两个count不会冲突，在两个名字空间里
    }
};
```

在使用名称空间中定义的标识符时，必须指定名称空间的名称，例如:

```
    my_space::count;
    my_space::print();
    my_space::inner::count;
```

为了避免程序中到处使用命名空间的名称，使用该语句:using namespace my_space;

则上述中的使用见简写为:

```
my_space::count;
print();
inner::count;
```

标准命名空间std:使用此命名空间，你可以写系统头文件没有扩展，如:

```
#include <iostream>
#include <string>
using namespace std;
```

【例1-1】命名空间的使用

```
#include <iostream>
using namespace std;//命名空间的使用

//定一个一个命名空间
namespace my_space{
    int count;
    const int size = 2;
    const double pi = 3.14;
    void print( );

    //命名空间的定义是可以嵌套的
    namespace inner{
        int count; //两个count不会冲突，在两个名字空间里
    }
};
void my_space::print( ){
    //如果不写using namespace std，此处就要写成std::cout
    cout << "Name space demo:\n";
    cout << "size = " << size << endl;
    cout << "pi = " << pi << endl;
    cout << "Inner::count = " << inner::count<<endl;
}
```

```
int main(int argc, const char * argv[]) {

    const int size = 2;
    double table[my_space::size][size];

    for(my_space::count = 0; my_space::count < my_space::size; ++
my_space::count){

        for(my_space::inner::count = 0; my_space::inner::count <
size; ++my_space::inner::count){

            //给二维数组的每一个元素赋值
            table[my_space::count][ my_space::inner::count] =
my_space::pi * my_space::count * my_space::inner::count;

            cout << "当前下标:" <<my_space::count <<"
"<<my_space::inner::count << endl;
        }
    }

    my_space::print( );

    return 0;
}
```

程序运行结果如下:

```
当前下标:0  0
当前下标:0  1
当前下标:1  0
当前下标:1  1
Name space demo:
size = 2
pi = 3.14
Inner::count = 2
```

1.2 运行时类型识别

运行时类型信息是由操作符：dynamic_cast。dynamic_cast用于向下转型，即基类的指针指向派生类。

（1）static_cast：静态转型(向下转型)。不管父类指针是否真的指向子类对象，一律把指针转型后返回。转型不保证安全。

（2）dynamic_cast：动态转型操作符。把父类指针转型成子类指针，如果转型成功（即父类指针的确指向子类对象），返回子类对象地址；如果转型失败，返回null。即转型是安全的。

使用格式:dynamic_cast<子类指针>(父类指针)，用来检测父类指针是否可以转换成子类指针。

【例1-2】动态类型转换的使用

```
/** Base.h  **/
#ifndef Base_h
#define Base_h

#include <iostream>
using namespace std;

class Base {
public:
    virtual void print()const;
    void outB()const;
};
#endif /* Base_h */

/** Base.cpp  **/
#include "Base.h"
void Base::print( ) const{
    cout<<"Base.\n"<<endl;
}

void Base::outB( ) const{
    cout<<"Out Base.\n"<<endl;
}
```

```
/** D1.h  **/
#ifndef D1_h
#define D1_h

#include "Base.h"

class D1 :public Base {
public:
    void print()const;
    void outD1()const;
};
#endif /* D1_h */

/** D1.cpp  **/
#include "D1.h"

void D1::print( ) const{
    cout << "D1.\n"<<endl;
}

void D1::outD1( ) const{
    cout << "Out D1.\n"<<endl;
}
```

```
/** D2.h  **/
#ifndef D2_h
#define D2_h

#include "Base.h"

class D2 :public Base {
public:
    void print()const;
    void outD2()const;
};
#endif /* D2_h */

/** D2.cpp  **/
#include "D2.h"

void D2::print( ) const{
    cout << "D2.\n"<<endl;
}

void D2::outD2( ) const{
    cout << "Out D2.\n"<<endl;
}
```

```
/** main.cpp  **/
#include <iostream>
#include "D1.h"
#include "D2.h"
int main(int argc, const char * argv[]) {
    D1 d1;
    D2 d2;
    const D1* p1 = NULL;
    const D2* p2 = NULL;
    Base* p[2] = {&d1, &d2}; //向上转型:存储的指针是父类指针
    for (int count = 0; count < 2; ++count){
        //向下转型:如果成功就是D1*,否则就是nil
        if ((p1 = dynamic_cast<D1*>(p[count]))){
            p1 -> print( );//调用子类对象d1的print函数
            p1 -> outD1( );
        } else if ((p2 = dynamic_cast<D2*>(p[count]))){

            p2 -> print( );//调用子类对象d2的print函数
            p2 -> outD2( );
        }
    }
    return 0;
}
```

程序运行结果如下:

```
D1.
Out D1.
D2.
Out D2.
```

注意: dynamic_cast转换的时候必须有virtual函数支持。

二、字符串类

我们目前使用的字符串是字符数组，以一个空字节结束。ANSI C++标准引入了一个内置的类称为字符串。这个类中的对象表示字符串，可以使用比C字符串更容易操作的字符串。

事实上，这类字符串的定义是一个模板类basic_string，使字符串类可以有其他成员函数来操作字符符。

使用字符串类必须包含头文件#include<string>和使用命名空间using namespace std。

(1) 这个类有一个默认构造函数来初始化一个字符串为空字符串。

```
string str;
```

(2) 它也有一个构造函数，接受一个字符串或一个字符的参数，用于创建一个字符串对象。

```
string str("I am student");
string str('i');
```

（3）构造函数可以使用一个隐式转换构造函数。

```
string str = "I am student!";
```

（4）类字符串还具有拷贝构造函数和赋值操作符重载。当一个字符串对象被分配到另一个字符串对象时，后者是前者的副本，并且它们是独立的对象。运算符<<可用于输出字符串对象。

```
string str = "I am student!";
cout<<str<<endl;
```

（5）字符串对象的长度是由成员函数length()来获取，长度返回字符串中的字符数。

```
string str = "I am student!";
cout<<str.length()<<endl;
```

（6）String类有一个重载操作符[ ]（int i），返回第i个字符的字符串，其中下标的范围从0到length( )-1长度。在一个字符串的第i个字符也可以由成员函数at（int index）此函数检查，看看是否在合法范围内。如果该索引是在边界之外，它抛出一个异常，而操作符[]没有检查的范围，虽然它是更直观的使用。

```
string str = "I am student!";
str.at(0) = 'M';
str[1] = ',';
cout<<str<<endl;
```

(7) 运算符+重载做字符串对象连接。我们还可以使用运算符+=连接字符串。

```
string str = "I am";
string str1 = " student";
string str2 = str+str1;
cout<<str2<<endl;
str = str+str1;
cout<<str<<endl;
```

(8) 字符串对象可以使用重载比较运算符 ==, !=, >, >=, <, or <=.

【例2-1】类和对象的使用

```
#include <iostream>
#include <string>
using namespace std;

//交换字符串
void swap(string& s1, string & s2){
    string temp = s1;
    s1 = s2;
    s2 = temp;
}
int main(int argc, const char * argv[]) {
    string s1("string");//调用字符串的构造函数
    string s2(s1);      //拷贝构造函数
    string  s3 ;        //默认构造函数，无参

    s3 = s2;            //重载赋值操作符
    cout << s1 + " " + s2 + " " + s3 + '\n'; //字符串相加
    s1 + " ";//这个加法运算完成的前提：加号两边类型一致；

    s2 = "another string";
    cout << "The length of s2 is:" << s2.length( ) << endl;
```

```
    s1[3] = 'o';     //非const版本下标操作符
    s2.at(8) = 'S'; //s2[8] = ‘S’;
    if (s1 > s2) {
        swap(s1, s2);
    }
    if (s2 > s3) {
        swap(s2, s3);
    }
    s1 += s2;
    s1 += s3;

    const char *p = s1.c_str();//c_str()这是一个封装好的函数，返回的时C的
字符串
    cout << p << endl;

    p = s1.data();
    cout << p << endl;

    return 0;
}
```

程序运行结果如下:

```
string string string
The length of s2 is 14
another Stringstringstrong
another Stringstringstrong
```

三、标准模版库(STL)

3.1 标准模版库(STL)简介

标准模板库（STL）是ANSI/ISO C++标准草案的重要组成部分。它已经被大多数主要的编译器实现并被广泛使用。

3.2 容器

容器是一种数据结构，用于存储与同类型的数据项的集合。

STL容器使用类模板实现。每个类都有特定的成员函数，用于处理这个类中的对象。

1、顺序容器

(1) 向量Vector：随机访问任何一个元素，尾部增删元素。

(2) 双端对列Deque： 随机访问，在头部和尾部增删元素。

(3) 链表List： 顺序访问，任意位置增删元素。

Vector容器，在头部增加、删除元素，其时间消耗与元素数目成正比,若加在尾部，消耗时间是一个常量。

对于Deque，在头部、尾部增加删除元素的时间，都是常量。

Vector和Deque重载了下标操作符，而list没有。所以Vector和Deque用[]来进行随机访问。

2、关联容器

(1) 集合Set：集合存储不能有重复元素。

(2) 多集合MultiSet：集合存储可以有重复元素。

(3) 映射Map：一对一的键值对(key,value)，值可以通过指定的键检索，其中键(key)不能重复

(4) 多映射MultiMap：一对多的键值对(key,value)，其中键(key)可以重复

要使用容器类，包括下面的头文件（在命名空间中定义的）<vector>，<list>，< deque >，<map>（对于map和multimap），<set>（为set和multiset）。

⑴ 每个容器类有默认和拷贝构造函数，析构函数，重载赋值运算符。

⑵ 判断容器是否为空empty()函数,函数返回布尔值。

⑶ 函数max_size()返回容器的最大存储空间。

⑷ 函数capacity()返回容器当前分配的空间。

⑸ 函数size()返回容器当前元素个数和比较运算符。

⑹ 容器还具有删除earse()的函数，该函数可以从容器中删除一个或多个元素，并清除容器中的所有元素clear()函数。

四、迭代器

使用迭代器，对容器进行元素枚举，使用迭代器也定义在头文件< iterator >

一个迭代器被用来“指向”容器中的元素（虽然不是指针）。迭代器也可以解引用(*)和进行算术运算。

迭代器可以声明如下格式:

(1) iterator: 前向遍历，可读可写
(2) const_iterator: 前向遍历，只读
(3) reverse_iterator: 后向遍历，可读可写
(4) const_reverse_iterator: 后向遍历，只读

迭代器函数:

(1) 函数began()返回一个迭代器指向容器的第一个元素。
(2) 函数end()返回一个迭代器指向容量的最后一个元素的下一位置后。
(3) 函数rbegin()返回一个迭代器指向容量的最后一个元素。
(4) 函数rend()返回一个迭代器指向容量的第一个元素之前的位置。

五 算法

STL也包含了一些“独立”的功能（称为算法），可以用来操作容器。

例如，一个算法find()找到一个给定的键的元素。如果该元素被发现，它返回该元素的一个迭代器；否则，它返回到该位置后的最后一个元素的位置。

另一种算法copy()将容器（或其一部分）的内容复制到另一个容器中。使用该算法，需要包括头文件<algorithm>。

六、STL示例代码

6.1 Vector容器

模板向量的一些其他成员函数:

(1) 重载操作符[]返回容器中某一个元素，参考两个版本：const和非const

(2) 函数frond()和back()返回第一个和最后一个元素

(3) 函数insert() 在一个给定的位置插入到一个新的元素

(4) 功能push_back()和pop_back()添加或删除最后一个成员

【例6-1】Vector容器的使用

```
#include <iostream>
#include <vector>      //容器
#include <algorithm>   //算法
#include <iterator>    //迭代器
using namespace std;

int main(int argc, const char * argv[]) {

    const int SIZE = 6;
    int array[SIZE] = {1, 2, 3, 4, 5, 6};

    //声明一个int类型集合的容器，并用数组a对容器进行初始化
    vector<int> v(array, array + SIZE);

    cout<<"First element:"<<v.front()<<"\nLast
element:"<<v.back()<<endl;

    //通过下标操作符和at函数来修改容器中元素内容
    //注意二者区别，后者更安全，at函数会检查下标是否越界。
    v[1] = 7;
    v.at(2) = 10;
```

```
//在第二个元素位置插入22元素
v.insert(v.begin() + 1, 22);

//尾部添加元素19
v.push_back(19);

//声明一个迭代器(必须vector<int>::iterator)
vector<int>::iterator iter;
iter = v.begin();//迭代器指向容器中第一元素位置
while (iter!=v.end()) {
    cout<<*iter<<endl;//类似指针解引运算符
    iter++;//迭代器向后移动
}

//查找元素5（不是第5个）的位置，并返回其迭代器。
iter = find(v.begin( ), v.end( ), 22);
if (iter != v.end( )){
    cout << "location:"<<(iter-v.begin())<<endl;
}else{
    cout << "not found"<<endl;
}
```

```
    //系统最大容量
    cout<<"The max size of the vector is:"<<v.max_size( );
    //当前最大容量
    cout<<"The current capacity is:"<<v.capacity()<<endl;

    try{
        v.at(1) = 777;  //越界，抛异常
    }
    catch(out_of_range & e ){
        cout << "Exception: "<<e.what();
    }

    v.erase(v.begin());//清除第一个元素

    v.erase(v.begin( ), v.end( ));  //清除所有元素

    cout<<"After erase,vector v "<<(v.empty()?"is":"is not")<<"
empty"<<endl;
```

```
    //从开始位置把数组插入容器中
    v.insert(v.begin(), array, array+SIZE);
    cout<<"After insert,vector v "<<(v.empty()?"is":"is not")<<"
empty"<<endl;

    v.clear(); //清空容器
    cout<<"After clear,vector v "<<(v.empty()?"is":"is not")<<"
empty"<<endl;

    return 0;
}
```

【例6-2】list的使用示例

```
#include <iostream>
#include <list>
using namespace std;

typedef list<int> IntList;//重新定义类型名字

int main(int argc, const char * argv[]) {

    IntList lst;//list<int> lst;

    for( int i = 0; i < 5; ++i ){
        lst.push_front( i );//把值加在前面
        lst.push_back( i ); //把值加在后面
    }

    IntList::iterator itor = lst.begin(); //定义一个迭代器
    while( itor != lst.end()){ //for( int i=0; i<lst.size(); ++i)

        cout<<*itor++<<" ";
    }
    cout<<endl;
```

```
    lst.remove(0);//移除所有值为0的元素。
    itor = lst.begin();
    for( int i = 0; i < lst.size(); ++i){
        cout<<*itor++<<" ";
    }
    cout<<endl;

    return 0;
}
```

程序运行结果如下：

4 3 2 1 0 0 1 2 3 4

4 3 2 1 1 2 3 4

【例6-3】set的使用

```
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;

int main(int argc, const char * argv[]) {

    set <int> s; //声明一个不能重复元素集合
    s.insert(10);
    s.insert (-5);
    s.insert(123);
    s.insert(2);
    s.insert(44);
    s.insert(10); //10不能插入,重复(自动被忽略)

    set<int>::iterator here; //声明一个迭代器
    for (here = s.begin( ); here != s.end( ); ++here){
        cout << *here << " ";
    }
    cout << endl;
```

```
    int key;
    cout << "Enter an integer to search.\n";
    cin >> key;
    here = s.find(key); //查找指定值的元素，返回其迭代器
    if (here != s.end( )){
        cout << key << " is in this set"<<endl;
    }
    else{
        cout << key << " is not in this set"<<endl;
    }
    return 0;
}
```

程序运行结果如下：

```
-5 2 10 44 123
Enter an integer to search.
20
20 is not in this set
```

【例6-4】map的使用

```
/** String.h  **/
#ifndef String_h
#define String_h

#include <iostream>
#include <map>
using namespace std;
//创建类用来做map的key，需要重载下面函数
class Strings
{
    friend ostream& operator<<(ostream& out, const Strings& s);
public:
    Strings( );
    Strings(char * s);
    Strings( const Strings& s);
    ~Strings();
    Strings& operator=(Strings& s);
    bool operator<(const Strings& s) const;

private:
    char * str;
};
#endif /* String_h */
```

```cpp
/** String.cpp  **/

#include "String.h"

Strings::Strings( ) : str(new char[1]){
    str[0] = '\0';
}

Strings::Strings(char * s){
    if (!s){
        str = NULL;
    }else{
        str = new char[strlen(s) + 1];
        strcpy(str, s);
    }
}

Strings::Strings( const Strings& s){
    if (!s.str){
        str = NULL;
    }else{
        str = new char[strlen(s.str) + 1];
        strcpy(str, s.str);
    }
}
```

```
/** String.cpp  **/

Strings::~Strings(){
    if(str!=NULL){
        delete [] str;
        str = NULL;
    }
}

Strings& Strings::operator=(Strings& s){
    if ( this == &s ){ return *this;}
    delete []str;
    str = NULL;
    str = new char[strlen(s.str) + 1];
    strcpy(str, s.str);
    return *this;
}

bool operator==(const Strings& s) const{
    bool equal = false;
    if (!strcmp(str, s.str)){
        equal = true;
    }
    return equal;
}
```

```cpp
/** String.cpp  **/

bool Strings::operator<(const Strings& s) const{
    bool less = false;
    if (strcmp(str, s.str) < 0) {
        less = true;
    }
    return less;
}

ostream& operator<<(ostream& out, const Strings& s){
    out << s.str;
    return out;
}
```

```
/** main.cpp  **/

int main(int argc, const char * argv[]) {

    //定义一个映射key->String value->double(key 不能重复)
    map<Strings, double> m;

    //调用value_type类的构造函数，创建一个键值对对象,插入到map容器中
    m.insert(map<Strings, double>::value_type(Strings((char*)"pi"),
3.1416));
    m.insert(map<Strings, double>::value_type((char*)"e", 2.7183));
    m.insert(map<Strings, double>::value_type((char*)"zero", 0));

    //因为key已经存在，添加相同key被忽略
    m.insert(map<Strings, double>::value_type((char*)"e", 100));

    //通过下标插入心的键值对
    m[(char*)"one"] = 1.0;

    //修改key为pi的值
    m[Strings((char*)"pi")] = 1.0;
```

```
    map<Strings, double>::const_iterator  iter;
    for (iter = m.begin( ); iter != m.end( ); ++iter){
        cout << iter -> first << " " << iter -> second<< endl;
    }

    cout << "The value of pi is " << m.find(Strings((char*)"pi")) -
> second << endl;

    return 0;
}
```

程序运行结果如下：

```
e 2.7183
one 1
pi 1
zero 0
The value of pi is 1
```