STL用法

2021年7月28日 9:49

STL学习

1、什么是STL?

STL（Standard Template Library），即标准模板库，是一个具有工业强度的，高效的C++程序库。它被容纳于C++标准程序库（C++ Standard Library）中，是ANSI/ISO C++标准中最新的也是极具革命性的一部分。该库包含了诸多在计算机科学领域里常用的基本数据结构和基本算法。为广大C++程序员们提供了一个可扩展的应用框架，高度地体现了软件地可复用性。

STL的一个重要特点是数据结构和算法的分离。尽管这是个简单的概念，但这种分离却使得STL变得非常通用。例如，由于STL的sort()函数是完全通用的，你可以用它来操作几乎任何数据集合，包括链表、容器和数组。

STL的另一个重要特性是它不是面向对象的。为了具有足够通用性，STL主要依赖于模板，而不是封装，继承和虚函数（多态性）——OOP的三要素。你在STL中找不到任何明显的类继承关系。这好像是一种倒退，但这正好是使得STL的组件具有广泛通用性的底层特征。另外，由于STL是基于模板，内联函数的使用使得生成的代码短小高效。

从逻辑层次来看，在STL中体现了泛型化程序设计的思想，引入了诸多新的名词，比如像需求（requirements），概念（concept），模型（model），容器（container），算法（algorithmn），迭代子（iterator）等。与OOP（object-oriented programming）中的多态（polymorphism）一样，泛型也是一种软件的复用技术；

从实现层次看，整个STL是以一种类型参数化的方式实现的，这种方式基于一个在早先C++标准中没有出现的语言特性--模板（template）。

2、STL内容

STL中六大组件：

（1）容器（Container），是一种数据结构，如list，vector，deques等，以模板类的方法提供。为了访问容器中的数据，可以使用由容器类输出的迭代器；

（2）迭代器（Iterator），提供了访问容器中对象的方法。例如，可以使用一对迭代器指定list或vector中的一定范围的对象。迭代器就如同一个指针。事实上，C++的指针也是一种迭代器。但是，迭代器也可以是那些定义了operator*()以及其他类似于指针的操作符的方法的类对象；

（3）算法（Algorithm）， 是用来操作容器中的数据的模板函数。例如，STL用sort()来对一个vector中的数据进行排序，用find()来搜索一个list中的对象，函数本身与他们操作的数据的结构和类型无关，因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用；

（4）仿函数（Functor）， 行为类似函数，可作为算法的某种策略。从实现角度来看，仿函数是一种重载了operator()的class或者class template。

（5）适配器（Adaptor）， 一种用来修饰容器或者仿函数或迭代器接口的东西。；

（6）空间配置器（Allocator）， 负责空间的配置与管理。从实现角度看，配置器是一个实现了动态空间配置、空间管理、空间释放的class tempalte。

2.1容器

STL中的容器有顺序容器、关联容器以及容器适配器。

（1）顺序容器： 是一种各元素之间有顺序关系的线性表，是一种线性结构的可序群集。顺序性容器中的每个元素均有固定的位置，除非用删除或插入的操作改变这个位置。顺序容器的元素排列次序与元素值无关，而是由元素添加到容器里的次序决定。顺序容器包括：vector(向量)、list（列表）、deque（队列）。

（2）关联容器：关联式容器是非线性的树结构，更准确的说是二叉树结构。各元素之间没有严格的物理上的顺序关系，也就是说元素在容器中并没有保存元素置入容器时的逻辑顺序。但是关联式容器提供了另一种根据元素特点排序的功能，这样迭代器就能根据元素的特点“顺序地”获取元素。元素是有序的集合，默认在插入的时候按升序排列。关联容器包括：

map（集合）、set（映射）、multimap（多重集合）、multiset（多重映射）。

（3）容器适配器：本质上，适配器是使一种不同的行为类似于另一事物的行为的一种机制。容器适配器让一种已存在的容器

类型采用另一种不同的抽象类型的工作方式实现。适配器是容器的接口，它本身不能直接保存元素，它保存元素的机制是调用另一种顺序容器去实现，即可以把适配器看作“它保存一个容器，这个容器再保存所有元素”。STL 中包含三种适配器：栈stack 、队列queue 和优先级队列priority_queue 。

2.2迭代器

Iterator（迭代器）模式又称Cursor（游标）模式，用于提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。或者这样说可能更容易理解：Iterator模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由iterator提供的方法）访问聚合对象中的各个元素。
迭代器的作用：能够让迭代器与算法不干扰的相互发展，最后又能无间隙的粘合起来，重载了*，＋＋，＝＝，！＝，＝运算符。用以操作复杂的数据结构，容器提供迭代器，算法使用迭代器；常见的一些迭代器类型：
iterator、const_iterator、reverse_iterator和const_reverse_iterator。

2.3算法

2.3.1说明

函数库对数据类型的选择对其可重用性起着至关重要的作用。举例来说，一个求方根的函数，在使用浮点数作为其参数类型的情况下的可重用性肯定比使用整型作为它的参数类性要高。而C++通过模板的机制允许推迟对某些类型的选择，直到真正想使用模板或者说对模板进行特化的时候，STL就利用了这一点提供了相当多的有用算法。它是在一个有效的框架中完成这些算法的——你可以将所有的类型划分为少数的几类，然后就可以在模版的参数中使用一种类型替换掉同一种类中的其他类型。
STL提供了大约100个实现算法的模版函数，比如算法for_each将为指定序列中的每一个元素调用指定的函数，stable_sort以你所指定的规则对序列进行稳定性排序等等。只要我们熟悉了STL之后，许多代码可以被大大的化简，只需要通过调用一两个算法模板，就可以完成所需要的功能并大大地提升效率。
算法部分主要由头文件<algorithm>，<numeric>和<functional>组成。
<algorithm>是所有STL头文件中最大的一个（尽管它很好理解），它是由一大堆模版函数组成的，可以认为每个函数在很大程度上都是独立的，其中常用到的功能范围涉及到比较、交换、查找、遍历操作、复制、修改、移除、反转、排序、合并等等。
<numeric>体积很小，只包括几个在序列上面进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。
<functional>中则定义了一些模板类，用以声明函数对象。

2.3.2算法分类

STL中算法大致分为四类：
（1）非可变序列算法：指不直接修改其所操作的容器内容的算法。
（2）可变序列算法：指可以修改它们所操作的容器内容的算法。
（3）排序算法：对序列进行排序和合并的算法、搜索算法以及有序序列上的集合操作。
（4）数值算法：对容器内容进行数值计算。

2.4仿函数

2.4.1说明

仿函数（Functor），就是使一个类的使用看上去像一个函数。其实现就是类中实现一个operator()，这个类就有了类似函数的行为，就是一个仿函数类了。
仿函数，又或叫做函数对象，是STL六大组件之一；仿函数虽然小，但却极大的拓展了算法的功能，几乎所有的算法都有仿函数版本。例如，查找算法find_if就是对find算法的扩展，标准的查找是两个元素相等就找到了，但是什么是相等在不同情况下却需要不同的定义，如地址相等，地址和邮编都相等，虽然这些相等的定义在变，但算法本身却不需要改变，这都多亏了仿函数。仿函数(functor)又称之为函数对象（function object），其实就是重载了()操作符的struct，没有什么特别的地方。

2.4.2仿函数在编程语言中的应用

（1）C语言中使用函数指针和回调函数来实现仿函数，例如一个用来排序的函数可以这样使用仿函数。

```
#include <stdio.h>
#include <stdlib.h>
//int sort_function( const void *a, const void *b);
int sort_function( const void *a, const void *b)
{
    return *(int*)a-*(int*)b;
}
int main()
{
   int list[5] = { 54, 21, 11, 67, 22 };
   qsort((void *)list, 5, sizeof(list[0]), sort_function);//起始地址，个数，元素大小，回调函数
   int  x;
   for (x = 0; x < 5; x++)
          printf("%i\n", list[x]);
   return 0;
}
```

（2） 在C++里，我们通过在一个类中重载括号运算符的方法使用一个函数对象而不是一个普通函数。

```
#include <iostream>
#include <algorithm>
using namespace std;
template<typename T>
class display
{
public:
    void operator()(const T &x)
    {
        cout << x << " ";
    }
};
int main()
{
    int ia[] = { 1,2,3,4,5 };
    for_each(ia, ia + 5, display<int>());
    system("pause");
    return 0;
}
```

2.4.3仿函数在STL中的定义

要使用STL内建的仿函数，必须包含<functional>头文件，而头文件中包含的仿函数分类包括：

（1）算术类仿函数

加：plus<T>

减：minus<T>

乘： multiplies<T>

除：divides<T>

模取：modulus<T>

否定：negate<T>

```
#include <iostream>
#include <numeric>
#include <vector>
#include <functional>
using namespace std;
int main()
{
    int ia[] = { 1,2,3,4,5 };
    vector<int> iv(ia, ia + 5);
    //120
    cout << accumulate(iv.begin(), iv.end(), 1, multiplies<int>()) << endl;
    //15
    cout << multiplies<int>()(3, 5) << endl;
    modulus<int>  modulusObj;
    cout << modulusObj(3, 5) << endl; // 3
    system("pause");
    return 0;
}
```

(2)关系运算类仿函数

等于：equal_to<T>

不等于：not_equal_to<T>

大于：greater<T>

大于等于：greater_equal<T>

小于：less<T>

小于等于：less_equal<T>

```
#include <iostream>
#include <algorithm>
```

```
#include<functional>
#include <vector>
using namespace std;
template <class T>
class display
{
public:
    void operator()(const T &x)
    {
        cout << x << " ";
    }
};
int main()
{
    int ia[] = { 1,5,4,3,2 };
    vector<int> iv(ia, ia + 5);
    sort(iv.begin(), iv.end(), greater<int>());
    for_each(iv.begin(), iv.end(), display<int>());
    system("pause");
    return 0;
}
```

(3)逻辑运算仿函数

逻辑与：logical_and<T>

逻辑或：logical_or<T>

逻辑否：logical_no<T>

除了使用STL内建的仿函数，还可使用自定义的仿函数。

2.4.3仿函数操作实例

（1）仿函数当作排序准则

```
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
using namespace std;
class Person{
    public:
        string firstname() const;
        string lastname() const;
        ...
};
class PersonSortCriterion{
    public:
        bool operator()(const Person&p1,const Person& p2) const {
            return p1.lastname()<p2.lastname()||
                    (!(p2.lastname()<p1.lastname())&&
                    p1.firstname()<p2.firstname());
        }
};
int main()
{
    typedef set<Person,PersonSortCriterion> PersonSet;
    PersonSet coll;
    PersonSet::iterator pos;
    for(pos=coll.begin();pos!=coll.end();++pos){
        ...
    }
    ...
}
```

这里的coll适用了特殊排序准则PersonSortCritersion，而它是一个仿函数类别。所以可以当做set的template参数，而一般函数则无法做到这一点。

（2）拥有内部状态的仿函数

```
#include <iostream>
#include <list>
#include <algorithm>
#include "print.cpp"
using namespace std;
class IntSequence
{
private:
    int value;
public:
    IntSequence(int initialValue):value(initialValue){}
    int operator() ()
    {
        return value++;
    }
};
int main()
{
    list<int> coll;
    generate_n(back_inserter(coll),9,IntSequence(1));
    PRINT_ELEMENTS(coll);
    generate(++coll.begin(),--coll.end(),IntSequence(42));
```

```
    PRINT_ELEMENTS(coll);
}
```

（3）for_each()的返回值

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class MeanValue
{
private:
    long num;
    long sum;
public:
    MeanValue():num(0),sum(0){}
    void operator() (int elem)
    {
        num++;
        sum+=elem;
    }
    double value()
    {
        return static_cast<double>(sum)/static_cast<double>(num);
    }
};
int main()
{
    vector<int> coll;
    for(int i=1;i<=8;++i)
    {
        coll.push_back(i);
    }
    MeanValue mv=for_each(coll.begin(),coll.end(),MeanValue());
    cout<<"mean value:"<<mv.value()<<endl;
}
```

2.5适配器

2.5.1说明

适配器，在STL中扮演者转换器的角色，本质上是一种设计模式，用于将一种接口转换成另一种接口，从而使原本不兼容的接口能够很好地一起运作。

2.5.2分类

根据目标接口的类型，可以分为三类：改变容器的接口，称为容器适配器；改变迭代器的接口，称为迭代器适配器；改变仿函数的接口，称为仿函数适配器。

（1）容器适配器

将基本容器转换为特殊容器。

- stack栈
- queue队列
- priority_queue优先队列

（2）迭代器适配器

- 反向迭代器（reverse_iterator）
- 安插型迭代器（inserter或者insert_iterator）
- 流迭代器（istream_iterator / ostream_iterator）
- 流缓冲区迭代器（istreambuf_iterator / ostreambuf_iterator）
- 移动迭代器（move_iterator）

（3）仿函数适配器

- bind2nd
- bind1st

- not1()

2.6空间配置器

空间配置器，顾名思义就是为各个容器高效地管理空间（空间地申请与收回）默默工作。

在使用vector、list、map、unordered_map等容器时，所有需要空间的地方都是通过new申请的，虽然代码可以正常运行，但是有以下不足之处：

- 空间申请与释放需要用户自己管理，容易造成内存泄漏；
- 频繁向系统申请小块内存块，容易造成内存碎片；
- 频繁向系统申请小块内存，影响程序运行效率；
- 直接使用malloc与new进行申请，每块空间前有额外空间浪费；
- 申请空间失败的应对问题；
- 代码结构比较混乱，代码复用率不高；
- 未考虑线程安全问题。

以上提到的几点不足之处，最主要还是：频繁向系统申请小块内存造成的。SGI-STL以128作为小块内存与大块内存的分界线，将空间配置器其分为两级结构，一级空间配置器处理大块内存，二级空间配置器处理小块内存。

- 一级空间配置器原理非常简单，直接对malloc与free进行了封装，并增加了C++中set_new_handle思想。
- 二级空间配置器专门负责处理小于128字节的小块内存。SGI-STL采用了内存池的技术来提高申请空间的速度以及减少额外空间的浪费，采用哈希桶的方式来提高用户获取空间的速度与高效管理。

来自 <https://blog.csdn.net/u010183728/article/details/81913729>