C++的标准库

C++ 标准函数库

STL (Standard Template Library)

1. C++ 标准库 (1)

C++语言规范提供了一套标准的函数库，主要由3部分组成：

- C函数库
- IO流及本地化
- STL（Standard Template Library）

  STL由类模板和函数模板组成，约占C++标准库的80%。

1. C++ 标准库 (2)

- C++标准库的所有头文件都没有扩展名。
- C++标准库包含50个头文件，其中18个提供了C库的功能。
- 在C++标准库中，与宏相关的名称(如max等)在全局作用域中定义，其他名称则在std命名空间中声明。
- 在C++中还可以使用C库的头文件（带.h后缀）。

2. STL介绍 (1)

- STL (Standard Template Library)，由类模板和函数模板组成，是C++标准库的子集，约占了C++标准库的80%。
- 在C++标准库中，STL被组织为13个头文件：<algorithm> 、 <deque> 、 <functional> 、<iterator> 、 <vector> 、 <list> 、 <map> 、<memory>、<numeric>、<queue>、<set>、<stack>、<utility>。

2. STL介绍 (2)

- STL包含6大组件：
  - 容器 (Container)
  - 迭代器 (Iterator)
  - 算法 (Algorithm)
  - 仿函数 (Functor)
  - 适配器 (Adaptor)
  - 内存分配器 (Allocator)

2.1 容器 (1)

- 容器是一种数据结构，用来存贮数据。如 list、vector、deques 等，以类模板的方法提供。为了访问容器中的数据，可以使用容器类中定义的迭代器。
- STL 提供 3 类标准容器：序列容器、排序容器、哈希容器。后两类容器有时也统称为关联容器。

2.1 容器 (2)

容器类别	描述
序列容器	包括 vector（向量）、list（列表）、deque（双端队列）。元素在容器中的位置同元素的值无关，即容器不是排序的。将元素插入容器时，指定在什么位置，元素就会位于什么位置。
排序容器	包括 set（集合）、multiset（多重集合）、map（映射）、multimap（多重映射）。排序容器中的元素默认是由小到大排序好的，即便是插入元素，元素也会插入到适当位置。
哈希容器	C++11 新加入 4 种关联式容器：unordered_set（哈希集合）、unordered_multiset（哈希多重集合）、unordered_map（哈希映射）、unordered_multimap（哈希多重映射）。哈希容器中的元素是未排序的，元素的位置由哈希函数确定。

2.2 迭代器 (1)

- 迭代器提供一种方法，顺序访问一个聚合对象中各个元素，不需暴露该对象的内部表示。每种容器类都会定义自己的迭代器。
- 迭代器就如同一个指针。事实上，C++的指针也是一种迭代器。
- 迭代器也可以是那些定义了operator*()以及其他类似于指针操作的类对象。
- 迭代器重载了 *，++，--，==，！=， = 运算符。

2.2 迭代器 (2)

迭代器可分2类：

- 容器迭代器：访问容器中的数据
- 流迭代器：访问流数据

迭代器可实现为：正向访问、双向访问、随机访问。

- 流迭代器 一般实现为正向迭代器。
- 容器迭代器 一般具有双向访问功能，而有些流迭代器还具有随机访问功能。

2.2 迭代器 (3)

- **正向迭代器**

  假设p是一个正向迭代器 (每次正向移动一个元素)，则p支持以下操作：++p，p++，*p。两个正向迭代器可以互相赋值，还可以用 == 和 != 运算符进行比较。

- **双向迭代器**

  双向迭代器除了具有正向迭代器的功能外，还具有-- (前置和后置) 功能 (每次正向或反向移动一个元素)。若p是一个双向迭代器，则 ++p、p++、--p、p--、==、!= 都可以用。

2.2 迭代器 (4)

随机访问迭代器 (1)

随机访问迭代器具有双向迭代器的全部功能，同时还具有随机访问的功能（每次正向或反向移动若干个元素)。若p是一个随机访问迭代器，i 是一个整型变量或常量，则：

++p、p++	向前移动1个元素（改变p）
--p、p--	向后移动1个元素（改变p）
p += i	p 往后移动 i 个元素（改变p）
p -= i	p 往前移动 i 个元素（改变p）
p + i	返回 p 后面第 i 个元素的迭代器（不改变p）
p – i	返回 p 前面第 i 个元素的迭代器（不改变p）
p[i]	返回 p 后面第 i 个元素的引用 (不改变p), 等价 *(p + i)

2.2 迭代器 (5)

随机访问迭代器 (2)

- 2个随机访问迭代器 p1、p2 还可以用 <、>、<=、>= 运算符。p1<p2 表示p2的位于p1之后，即p1需要经过若干次（至少1次）++操作后，才会等于 p2。p2-p1表示 p2 和 p1所指位置之间的元素个数。

- 不是所有容器都支持上面这些迭代器。stack、queue 和 priority_queue 没有迭代器，但这些容器适配器提供一些成员函数去访问容器中的元素。

容器	迭代器功能
vector	随机访问
deque	随机访问
list	双向
set / multiset	双向
map / multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

2.2 迭代器 (6)

(1) 容器迭代器

表1. 常用的容器迭代器
(常量迭代器指向的元素具有const属性)

迭代器类型	迭代器定义方法
正向迭代器	容器类名::iterator 迭代器名;
常量正向迭代器	容器类名::const_iterator 迭代器名;
反向迭代器	容器类名::reverse_iterator 迭代器名;
常量反向迭代器	容器类名::const_reverse_iterator 迭代器名;

2.2 迭代器 (7)

对于迭代器的 ++ 和 --运算，一般使用前置方式 (++i)，而不建议使用后置方式 (i++)，这是因为后置运算需要多生成一个临时的局部对象。

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v;
    for(int n = 1; n <= 5; n++) {
        v.push_back(n);          //在容器v的尾部添加一个元素
    }
    vector<int>::iterator i;   //定义正向迭代器
    for(i = v.begin(); i != v.end(); ++i) {    //用迭代器遍历容器
        cout << *i << " ";       //1 2 3 4 5
        *i *= 10;                //每个元素*10
    }
    //用反向迭代器遍历容器
    vector<int>::reverse_iterator j;
    for(j = v.rbegin(); j != v.rend(); ++j) {
        cout << *j << " ";       //50 40 30 20 10
    }
    return 0;
}
```

2.2 迭代器 (8)

(2) 流迭代器 (1)

输入流迭代器、输出流迭代器 (头文件：iterator)

输入流迭代器

istream_iterator <数据类型> 迭代器名(绑定的流=cin);

istream_iterator <数据类型> 迭代器名; //指向流的结束位置

输出流迭代器

ostream_iterator <数据类型> 迭代器名(绑定的流, 分隔符);

支持的操作

*p、++p、p++、==、!= 。

2.2 迭代器 (9)

(2) 流迭代器 (2)

```
#include<iostream>
#include<iterator>
#include<string>
using namespace std;
int main( ) {
    istream_iterator<string> in_iter(cin), eof;         //eof = Ctrl^Z
    ostream_iterator<string> out_iter(cout,"\n");  //元素分隔符为换行
    while(in_iter != eof) {
        string s = *in_iter;
        *out_iter = s;
        out_iter++;
        in_iter++;
   }
}
```

2.3 算法 (1)

- 算法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.STL提供了大约100个实现算法的模版函数，定义在3个头文件 <algorithm>、<numeric> 和 <functional> 中。
- <algorithm> 由很多模版函数组成的，可以认为每个函数在很大程度上都是独立的，比较常用的有：比较、交换、查找、遍历、复制、修改、移除、反转、排序、合并，等等。
- <numeric> 包括几个在序列上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 算法 (2)

- STL中算法大致分为四类：
  - 非可变序列算法：不直接修改容器的内容(元素)。
  - 可变序列算法：可以修改容器的内容。
  - 排序算法：对序列进行排序、合并、搜索。
  - 数值算法：对容器内容进行数值计算。

2.4 仿函数 (1)

- 仿函数就是一个实现了operator( )的类（这个类的对象就是函数对象），这样的类具有类似函数的行为（将类名当函数名一样使用）。
- C++在<functional>头文件中定义了如下三类仿函数：
  - 算术类仿函数： plus<T>、minus<T>、multiplies<T>、divides<T>、negate<T> 等
  - 关系运算类仿函数： equal_to<T>、not_equal_to<T>、greater_equal<T> 等
  - 逻辑运算仿函数： logical_and<T>、logical_or<T>、logical_not<T> 等
- 调用方式：multiplies<int>()(10, 123)，结果为1230.

2.4 仿函数 (2)


```
#include <iostream>
#include <algorithm>
using namespace std;
class Sort {
public:
      bool operator()(int x, int y) const {
      return x > y;
   }
};
class Display {
public:
     void operator()(int x) const { cout << x; }
};
int main( ) {
     int a[5] = { 4, 1, 2, 5 };
     sort(a, a+4, Sort());   //Sort()是函数名, sort()内部会调用 Sort()(v1,v2)
     for_each(a, a+4, Display());
     //for_each(p1, p2, Display())  <=>  while(p1 != p2) Display()(*p1++)
     return 0;
}
```

2.5 适配器 (1)

- 容器适配器对容器进行包装，使其表现出另外一种行为。
- C++标准库提供了3种顺序容器适配器：queue (FIFO队列)、priority_queue (优先级队列)、stack (栈)。
- 适配器对容器进行包装，就可实现其他相应的功能。例如，利用适配器 stack 去包装容器 vector<int> (即 stack<int, vector<int>>) 就实现了整型栈的功能。

2.5 适配器 (2)

華中科技大學

基础容器： 能够被适配器进行装配的容器，基础容器必须满足一定的条件。

容器适配器	基础容器筛选条件	默认使用的基础容器
stack	基础容器需包含以下成员函数： • empty() • size() • back() • push_back() • pop_back() 满足条件的基础容器有 vector、deque、list。	deque
queue	基础容器需包含以下成员函数： • empty() • size() • front() • back() • push_back() • pop_front() 满足条件的基础容器有 deque、list。	deque
priority_queue	基础容器需包含以下成员函数： • empty() • size() • front() • push_back() • pop_back() 满足条件的基础容器有vector、deque。	vector

2.6 内存分配器

内存分配器为容器类模板提供自定义的内存管理功能 (申请和释放)。一般情况下，只有高级用户才有改变内存分配策略的需求，因此内存分配器对于一般用户来说，并不常用。

2.7 程序示例 (1)


```
//程序 1：动态 1 维数组及排序
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> v = { 4, 2, 3, 1 };
    v.push_back(100);                //4, 2, 3, 1, 100
    sort(v.begin(), v.end());        //从小到大排序
    reverse(v.begin(), v.end());  //倒装数据
    for(int k = 0; k < v.size(); k++) {
        cout << v[k] << "    ";      //100 4 3 2 1
    }
    return 0;
}
```

```
//程序 2：动态 1 维数组及排序
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool compare(int x, int y)
{
        return x > y;    //降序排序
}

int main()
{
    vector<int> v = { 4, 2, 3, 1 };
    v.push_back(100);    //4, 2, 3, 1, 100
    sort(v.begin(), v.end(), compare);
    //reverse(v.begin(), v.end());
    for(int k = 0; k < v.size(); k++) {
            cout << v[k] << "    "; //100 4 3 2 1
    }
    return 0;
}
```

2.7 程序示例 (2)


```
//程序 3：动态 2 维数组
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int M = 3, N = 4;
    vector<vector<int>>
    v(M, vector<int>(N));
    int k = 1;
    for(int r = 0; r < v.size(); r++) {
        for(int c = 0; c < v[r].size(); c++) {
            v[r][c] = k++;
        }
    }
    return 0;
}
```

```
//程序 4：不规则 2 维数组
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int M = 4;
    vector<vector<int>> v(M);
    for(int r = 0; r < v.size(); r++) {
        v[r].resize(r+1);
    }
    int k = 1;
    for(int r = 0; r < v.size(); r++) {
        for(int c = 0; c < v[r].size(); c++) {
            v[r][c] = k++;
        }
    }
    for(int r = 0; r < v.size(); r++) {
        for(int c = 0; c < v[r].size(); c++) {
            cout << v[r][c] << "    ";
        }
        cout << "\n";      // 1
    }                       // 2 3
    return 0;               // 4 5 6
}                           // 7 8 9 10
```