STL — Standard Template Library

19 广电工 亓泽鲁

Tel : 18810959639

Based on 17 广电工 刘宗鑫

Menu

1. `Stack` & `Queue`
2. `Priority_queue`
3. `Vector`
4. `Map` & `Multimap`
5. `Set`
6. `Bitset`
7. `List`

§ `Stack` & `Queue`

`stack`和`queue`的知识我们在上学期的课程中已经学习了，本节课主要是给大家回顾一下两种容器的基本操作和用法。

`Stack`的头文件

```
#include <stack>
```

`Queue`的头文件

```
#include <queue>
```

§ 1.1 `Stack`

`Stack` 常用函数

函数	用法	作用
`empty()`	`StackName.empty()`	检测栈是否为空，空则返回真
`top()`	`StackName.top()`	返回栈顶元素
`pop()`	`StackName.pop()`	移除栈顶元素
`push()`	`StackName.push()`	在栈顶增加元素
`size()`	`StackName.`	返回栈中元素的数目

§ 1.2 Queue

函数	用法	作用
`empty()`	`QueueName.empty()`	检测队列是否为空，空则返回真
`push()`	`QueueName.push()`	在队尾插入一个元素
`pop()`	`QueueName.pop()`	弹出队首的元素
`size()`	`QueueName.size()`	返回队伍的元素数目
`front()`	`QueueName.front()`	返回队列中的第一个元素
`back()`	`QueueName.back()`	返回队尾元素

§ 2.1 `Priority_queue`

用途

保证队首元素优先级最大，**相当于大根堆。**

需要添加头文件：

```
#include <queue>
```

常用函数

- 基本和`queue`相同，不过优先队列保证队首元素优先级最大，而不是先进先出。
- 队首元素需要使用`top()`而不是`front()`

优先级的设置

需要重载<

但是，排序结果是相反的，因为他是按照优先级来的，优先级越大，会排在前边。

优先队列的默认优先级是降序的，但是如果需要设置小的数据优先级高，需要添加头文件：

```
#include <functional>
```

同时做以下修改：

```
priority_queue< int, vector<int>, greater<int> > q;
```

§ 3.1 `Vector`

`vector` 可以理解成**可变长的数组**，也就是数组的长度是可以动态变化的。

`vector` 常用于**需要存储内容大小不确定**，用普通数组存储会超内存的情况。

使用时需要添加头文件：

```
#include <vector>
```

基本操作

定义

基本

```
vector <type > name;
```

嵌套使用，可以建立**两个维度的都变化**的数组。

```
vector <vector <type > > name;
```

元素访问

1. 使用下标

   和普通数组一样，假设有一个 `vector` 变量名为 `v`，那么访问使用 `v[i]` 就可以访问第 `i` 个元素。

2. 使用迭代器

   正常的迭代器使用起来比较麻烦，在竞赛里推荐使用 `auto` 来实现。

```
vector <int > v;
for (auto i : v) cout << i << endl;
```

   但是需要注意的是，如果要修改其中的元素，那么就需要写成：

```
for (auto &i : v) {
    i = i + 1;
    cout << i << endl;
}
```

常用函数

`push_back`和`pop_back`

`push_back()`，就是再当前`vector`的后边添加一个元素，而`pop_back()`就是再`vector`后边删除一个元素。

`clear()`和`empty()`

`clear()`用于清空`STL`，`empty()`用于判断是否为空，如果为空返回1，否则返回0。

`insert`

`insert()` 有三种形式：

函数	作用
`v.insert(it, val)`	向迭代器 `it` 指向的元素前插入新元素 `val`
`v.insert(it, n, x)`	向迭代器 `it` 指向的元素前插入 `n` 个 `x`
`v.insert(it, first, last)`	将由迭代器 `first` 和 `last` 所指定的序列(`first` , `last`)插入到迭代器 `it` 指向的元素前面

`erase()`

函数	作用
`v.erase(it)`	删除由迭代器 `it` 所指向的元素
`v.erase(first, last)`	删除由迭代器 `first` 和 `last` 所指定的序列(`first` , `last`)

`vector`的一些个作用：

1. 邻接矩阵存储图（后面的课会有）
2. 用`vector`模拟`stack`， `stack`的速度比`vector`慢，当然更快的是直接用数组手模。

§ 4.1 `Map`

`map`翻译为映射。实际上数组也相当于映射，如`double a[100]`，则是建立100个从`int`到`double`的映射。但是数组如果要实现从字符串到`int`的映射，或者实现从一个结构体到另一个结构体的映射则就不那么方便了。而`map`就是为了解决这种情况而产生的。

需要添加头文件：

```
#include <map>
```

基本操作

定义

```
map <type1, type2 > MapName;
```

`map`理论上可以实现从任何类型到任何类型（废话）
但是！ 当`type1`要使用字符串类型的话，就必须使用`String`而不能使用`char`数组。因为数组不能时键，但是`type2`可以是数组类型。

元素访问

1. 通过键访问，$map[key_{value}] = value$
2. 推荐使用 `auto`

```
for (auto i : map_name) {
    //do sth
    cout << i.first << ' ' << i.second << endl;
}
```

常用函数

用法	功能
`mp[key] = x`	利用数组方式插入数据，`key`是键，`x`是值
`mp.size()`	返回`map`的大小
`mp.clear()`	清空`map`
`mp.empty()`	判断映射是否为空
`mp.count(key)`	存在`key`返回1否则为0

`map`的**键值对是唯一**的，如果如果出现重复的键，那么**后来者居上**，后来的键值对会替换之前的。

拓展

`map`的一些个用法：

1. 可以当作`bool`数组用
2. 可以实现离散化

时间复杂度：`map`的插入删除查找操作都是$O(logn)$的复杂度

§ 4.2 `Multimap`

使用时添加头文件：

```
#include <map>
```

它和`map`的不同在于，`multimap`的键值对不唯一，且只能通过迭代器访问，添加元素时只能使用`insert()`函数

§ 5.1 Set

译作**集合**，可以实现元素的自动去重和排序。

使用时添加头文件

```
#include <set>
```

定义：

```
set <type> Set_Name
```

基本操作

访问元素

只能通过迭代器!

```
#include <bits/stdc++.h>

using namespace std;

int main()
{
    set<int > s;
    s.insert(1);
    s.insert(2);
    s.insert(1);
    auto it = s.begin();
    cout << *(++ it) << " " << *( -- it ) << endl;
}
```

遍历元素

推荐使用`auto`

```
#include <bits/stdc++.h>

using namespace std;

int main()
{
    set<int > s;
    s.insert(1);
    s.insert(2);
    s.insert(1);
    for (auto i : s) cout << i << endl;
}
```

常用函数

用法	功能
`s.begin()`	返回指向第一个元素的迭代器
`s.end()`	返回指向最后一个元素的迭代器
`s.count(val)`	返回值为 `val` 的元素个数
`s.find()`	返回指向被查找到的元素的迭代器
`s.lower_bound(val)`	返回第一个大于等于 `val` 的元素迭代器
`s.upper_bound(val)`	返回第一个大于 `val` 的元素的迭代器
`s.size()`	返回集合中元素的个数

`set` 自带的 `lower_bound` 比 `algorithm` 快!

§ 6.1 Bitset

一种比较特殊的结构，他的元素**只能是0或者1。** 每一个元素只占1 `bit` 空间，使用它存储可以节省空间。

使用时添加头文件：

```
#include <bitset>
```

定义方法如下：

```
bitset<length > Name;
```

基本操作

元素访问

可以通过下标访问某一位，此时左边位为低位，右边位为高位。

```
#include <iostream>
#include <bitset>

using namespace std;

int main()
{
    bitset<8> bit(25);
    cout << bit[0] << endl;
}
//Output : 1
```

同时也支持直接输出全部二进制序列

```cpp
#include <iostream>
#include <bitset>

using namespace std;

int main()
{
    bitset<8> bit(25);
    cout << bit << endl;
}

//Output : 00011001
```

构造函数

用法	作用
`bitset<len> bit`	无参构造函数默认全0
`bitset<len> bit(string)`	将`string`转换成二进制，但是此时`string`必须是01串
`bitset<len> bit(char [])`	同上
`bitset<len> bit(val)`	将`val`转换成二进制存储

常用函数

用法	作用
`bit.size()`	返回位数
`bit.count()`	返回1的个数
`bit.reset()`	全部置0
`bit.set()`	全部置1
`bit.flip()`	全部取反

使用情况

- 进行二进制操作
- 可以节省空间

§ 7.1 `List`

其实就是链表，如果学过课内《数据结构》课的同学应该不会陌生。

使用使添加头文件：

```
#include <list>
```

定义时：

```
list <Type> ListName;
```

基本操作

访问元素

只能用迭代器访问

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list< int > l;
    l.push_back(1);
    auto it = l.begin();
    cout << *it << endl;
}
```

常用函数

用法	功能
`L.push_back()`	插入元素到链表尾部
`L.push_front()`	插入元素到链表首部
`L.front()`	返回链表首部元素
`L.back()`	返回链表尾部元素
`L.empty()`	判断链表是否为空

使用情况

- 支持快速删除插入操作
- 节约空间
- 模拟 `deque`

后记：STL的内容繁多且复杂，我们一节课的时间肯定无法做到面面俱到，需要大家在课下做题学习过程中不断积累。

Thanks For Your Listening