



# 论C++语言在信息学 竞赛中的应用

浙江省余姚中学 韩文弢

# 关于信息学竞赛

- 信息学竞赛一般要求在一定的时间内，理解并分析题意，设计符合给定时间和空间复杂度要求的算法，并在计算机上使用一定的程序设计语言正确地实现算法。
- 由于整个竞赛存在时间限制，因此所使用的程序设计语言能否正确、快速地实现算法对竞赛的成绩影响颇大。

# 关于信息学竞赛

- 所以，编程复杂度成为和算法的时间以及空间复杂度同等重要的因素。
- 编程复杂度在很大程度上与所选用的程序设计语言有关。
- 一般信息学竞赛中比较常用的程序设计语言有**BASIC**、**Pascal**、**C++**、**Java**等。

# 信息学竞赛中常用语言的特点

	BASIC	Pascal	C++	Java
学习难度	容易	一般	较难	较难
语言特点	简单	严谨	灵活	高度面向对象
运行速度	慢	较快	快	慢
库的功能	弱	一般	很强	强

# 中学信息学竞赛的语言现状

- BASIC语言正逐渐被淘汰。 ↘
- Pascal语言使用较为广泛，基本保持稳定。 →
- C++语言凭借其本身所具有的高度的灵活性，以及它所带的库的强大功能，被越来越多的选手所使用。 ↗

# 本文的目的和结构

- 目的：使读者在掌握**Pascal**语言的前提下，能尽快地掌握**C++**语言，并在此基础上进一步深入**C++**语言的高级应用。
- 结构：
  - 1 从**Pascal**到**C++**
  - 2 深入**C++**语言
  - 3 **STL**简介

# 3 STL简介

- 阅读本章的必要条件：了解C++面向对象程序设计的基础知识、了解一定的算法知识
- 本章的结构：
  - 3.1 STL概述
  - 3.2 迭代器
  - 3.3 算法
  - 3.4 容器
  - 3.5 本章小结



## 3.1 STL概述



# 一般化编程

## ■ 一般化编程(generic programming)的提出

```
□ void swap(int& x, int& y) {  
    int t = x;  
    x = y;  
    y = t;  
}
```

# 模板函数

## ■ 模板函数

```
□ template<class T>
  void swap(T& x, T& y) {
    T t = x;
    x = y;
    y = t;
  }
```

# 模板函数的调用

## ■ 模板函数的调用

### □ 隐式调用

- `swap(x, y);`

### □ 显式调用

- `swap<int>(x, y);`

# 模板类

## ■ 模板类

```
□ template<class T, int max>
  struct c_array {
    typedef T value_type;
    typedef T& reference;
    typedef const T& const_reference;
    T v[max];
    operator T* ();
    reference operator [] (size_t i);
    const_reference operator [] (size_t i) const;
    size_t size() const;
  };
```

# 模板类的使用

## ■ 模板类的使用

- `c_array<int, 100> a;`

- `c_array<double, 20> b;`

- `c_array<c_array<int, 10>, 10> c;`

# STL概述

- **STL**就是建立在模板函数和模板类基础之上的功能强大的库
  - 模板函数可以实现一般化的常用算法（如统计、排序、查找等）
  - 模板类可以实现支持几乎所有类型的容器，用来实现常用的数据结构（如链表、栈、队列、平衡二叉树等）

# STL头文件一览

头文件	内容	头文件	内容
<iterator>	迭代器	<vector>	向量
<utility>	辅助功能	<deque>	双头队列
<memory>	内存管理	<list>	链表
<algorithm>	算法	<set>	集合与多重集合
<functional>	函数对象	<map>	映射与多重映射
<numeric>	数值运算	<stack>	栈
		<queue>	队列与优先队列



## 3.2 迭代器



# 迭代器的定义和种类

- 迭代器(**iterator**)实际上是一种一般化的指针类型，是对指针类型的抽象。
- 根据所支持操作的不同，迭代器被分为五大类：
  - 输出迭代器(input iterator)
  - 输入迭代器(output iterator)
  - 前向迭代器(forward iterator)
  - 双向迭代器(bidirectional iterator)
  - 随机迭代器(random access iterator)

# 各种迭代器的功能

迭代器类型	输出迭代器	输入迭代器	前向迭代器	双向迭代器	随机迭代器
缩写	Out	In	For	Bi	Ran
读取	不支持	$x = *p$	$x = *p$	$x = *p$	$x = *p$
操作	不支持	$p \rightarrow x$	$p \rightarrow x$	$p \rightarrow x$	$p \rightarrow x$ $p[i]$
写入	$*p = x$	不支持	$*p = x$	$*p = x$	$*p = x$
迭代	$++$	$++$	$++$	$++$ $--$	$++$ $--$ $+$ $-$ $+=$ $-=$
比较	不支持	$==$ $!=$	$==$ $!=$	$==$ $!=$	$==$ $!=$ $<$ $>$ $<=$ $>=$

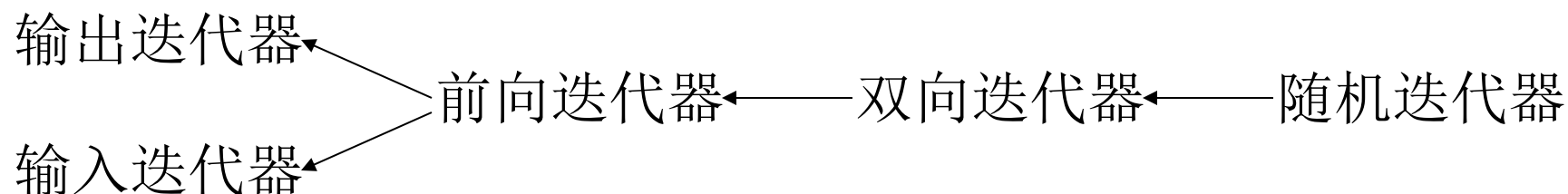


# 更多关于迭代器的信息

- 指针类型就是一种特殊的随机迭代器类型。
- 对于一般的迭代器，这些功能都是通过操作符重载来实现的。

# 更多关于迭代器的信息

## ■ 各种迭代器类型之间的关系：



## ■ 迭代器的作用

- 访问元素
- 算法与容器之间的纽带

# 模板类pair

```
■ template<class T1, class T2>
    struct pair {
        typedef T1 first_type;
        typedef T2 second_type;
        T1 first;
        T2 second;
        pair() : first(T1()), second(T2()) { }
        pair(const T1& x, const T2& y)
            : first(x), second(y) { }
        template<class U, class V>
        pair(const pair<U, V>& p)
            : first(p.first), second(p.second) { }
    };

```

# 模板类pair

```
■ template<class T1, class T2>
  pair<T1, T2>
  make_pair(const T1& x, const T2&
y) {
    return pair<T1, T2>(x, y);
}
```

## ■ 作用

- 储存一对密切相关的值
- 使用在当算法需要返回两个值时
- 作为映射的元素类型（关键字和被映射的值）



## 3.3 算法

# 与算法有关的知识

## ■ 算法(algorithm)

- 每个算法都是一个或者一组模板函数，用来完成一项特定的操作。

## ■ 序列(sequence)

- 序列用两个迭代器来描述，表示一组连续的元素；其中，第一个迭代器指向序列中的第一个元素，第二个迭代器指向序列最后一个元素的后一个位置。



# 与算法有关的知识

- 函数对象(function object)

- 函数对象重载了函数调用操作符(**operator ()**), 可以像普通函数一样被使用。

- 谓词(predicate)

- 返回值类型为**bool**的函数对象

# 函数对象举例

```
■ template<class T> class Sum {  
    T res;  
    public:  
        Sum(T i = T()) : res(i) { }  
        void operator () (const T& x) {  
            res += x;  
        }  
        T result const { return res; }  
};
```

# 常用函数对象

- STL在头文件 `<functional>` 中提供了一些常用运算的函数对象。

类名	类型	作用
<code>equal_to</code>	双目	<code>arg1 == arg2</code>
<code>not_equal_to</code>	双目	<code>arg1 != arg2</code>
<code>greater</code>	双目	<code>arg1 &gt; arg2</code>
<code>less</code>	双目	<code>arg1 &lt; arg2</code>
<code>greater_equal</code>	双目	<code>arg1 &gt;= arg2</code>
<code>less_equal</code>	双目	<code>arg1 &lt;= arg2</code>
<code>logical_and</code>	双目	<code>arg1 &amp;&amp; arg2</code>
<code>logical_or</code>	双目	<code>arg1    arg2</code>
<code>logical_not</code>	单目	<code>!arg</code>
<code>plus</code>	双目	<code>arg1 + arg2</code>
<code>minus</code>	双目	<code>arg1 - arg2</code>
<code>multiplies</code>	双目	<code>arg1 * arg2</code>
<code>divides</code>	双目	<code>arg1 / arg2</code>
<code>modulus</code>	双目	<code>arg1 % arg2</code>
<code>negate</code>	单目	<code>-arg</code>

# STL算法一览

## ■ 访问元素类

- `for_each()`, `transform()`

## ■ 顺序查找类

- `find()`, `find_if()`, `find_first_of()`, `adjacent_find()`,  
`search()`, `find_end()`, `search_n()`

## ■ 统计类

- `count()`, `count_if()`

# STL算法一览

## ■ 比较类

- mismatch(), equal(), lexicographical\_compare()

## ■ 复制类

- copy(), copy\_backward()

## ■ 交换类

- swap(), iter\_swap(), swap\_ranges()

# STL算法一览

## ■ 替换类

- `replace()`, `replace_if()`, `replace_copy()`,  
`replace_copy_if()`

## ■ 填充发生类

- `fill()`, `fill_n()`, `generate()`, `generate_n()`

## ■ 删除类

- `remove()`, `remove_if()`, `remove_copy()`,  
`remove_copy_if`

# STL算法一览

- 去重类

- `unique()`, `unique_copy()`

- 反转类

- `reverse()`, `reverse_copy()`

- 旋转类

- `rotate()`, `rotate_copy()`

- 随机打乱类

- `random_shuffle()`

# STL算法一览

## ■ 排序类

- `sort()`, `stable_sort()`, `partial_sort()`,  
`partial_sort_copy()`, `nth_element()`

## ■ 二分查找类

- `lower_bound()`, `upper_bound()`, `equal_range()`,  
`binary_search()`

## ■ 合并类

- `merge()`, `inplace_merge()`



# STL算法一览

## ■ 分区类

- `partition()`, `stable_partition()`

## ■ 集合运算类

- `includes()`, `set_union()`, `set_intersection()`,  
`set_difference()`, `set_symmetric_difference()`

## ■ 堆操作类

- `make_heap()`, `push_heap()`, `pop_heap()`,  
`sort_heap()`

# STL算法一览

- 最大最小类

- `min()`, `max()`, `min_element()`, `max_element()`

- 排列类

- `next_permutation()`, `prev_permutation()`

- 数值运算类

- `accumulate()`, `inner_product()`, `partial_sum()`,  
`adjacent_difference()`

# 常用算法介绍（排序）

## ■ 排序算法原型

```
□ template<class Ran>  
  void sort(Ran first, Ran last);  
  template<class Ran, class Cmp>  
    void sort(Ran first, Ran last, Cmp cmp);
```

## ■ 时间复杂度：平均 $O(n \log n)$ ，最坏 $O(n^2)$

## ■ 使用举例

```
□ sort(a, a + n);  
□ sort(b, b + m, greater<int>());
```

# 常用算法介绍（二分查找）

## ■ 二分查找算法原型：

- `template<class For, class T>`  
`bool binary_search(For first, For last, const T& val);`
- `template<class For, class T>`  
`For lower_bound(For first, For last, const T& val);`
- `template<class For, class T>`  
`For upper_bound(For first, For last, const T& val);`
- `template<class For, class T>`  
`pair<For, For> equal_range(For first, For last,`  
`const T& val);`

## ■ 时间复杂度：随机迭代器 $O(\log n)$ ，其他 $O(n)$

# 常用算法介绍（二分查找）

## ■ 算法的要求

- 序列有序
- 查找的谓词与排序的谓词相同

## ■ 算法的作用

- `binary_search()` 返回 `val` 是否在序列中。
- `lower_bound()` 返回指向序列中第一个大于或等于 `val` 的元素的迭代器。
- `upper_bound()` 返回指向序列中第一个大于 `val` 的元素的迭代器。
- `equal_range()` 返回一个 `pair`，表示序列中与 `val` 相等的元素所构成的子序列。

# 算法的组合使用

- 例如，要对一组范围较大的整数进行离散化，步骤如下：
  - 用`sort()`排序；
  - 用`unique()`去掉重复的元素；
  - 对于每个整数，用`lower_bound()`查找在序列中的位置。

# 例题：最长单调递增子序列

## ■ 题目描述

- 给定一个长度为 $n$ 的整数序列 $A = \{a_1, a_2, \dots, a_n\}$ , 求一个最大的整数 $m$ , 使得存在另一个序列 $P = \{p_1, p_2, \dots, p_m\}$ , 满足 $1 \leq p_1 < p_2 < \dots < p_m \leq n$  且  $a_{p_1} < a_{p_2} < \dots < a_{p_m}$ 。

## ■ 约束条件

- $n$ 不超过30,000
- $a_i$ 在 $[0, 1,000,000,00)$ 的区间内

# 思路分析

## ■ 原算法

- 设 $f_i$ 表示结尾元素为原序列中第 $i$ 个元素的最长单调递增序列的长度（为了简便，设 $a_0=-\infty$ ,  $f_0=0$ ），状态转移方程如下：

$$f_i = \max_{0 \leq j < i \wedge a_j < a_i} \{f_j + 1\}$$

- 时间复杂度 $O(n^2)$ ，不符合要求。







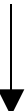









# 思路分析

## ■ 改进后的算法

- 设 $g_i$ 表示到目前为止，所有长度为 $i$ 的单调递增子序列中最后一个元素的最小值。
- 易知， $g_{i-1} \leq g_i$ 。
- 当到第 $i-1$ 个字符为止的 $\{g_i\}$ 已知时， $f_i$ 就等于在 $\{g_i\}$ 中第一个大于或等于 $a_i$ 的元素的位置 $j$ 。
- 此时，令 $g_j = a_i$ ，更新 $\{g_i\}$ 。
- 一开始， $g_i = \infty$ 。
- 时间复杂度 $O(n \log n)$ ，符合要求。

# 改进算法运行实例

$n=10$

										
$i$	1	2	3	4	5	6	7	8	9	10
$a_i$	3	1	4	1	5	9	2	6	5	3
$f_i$	1	1	2	1	3	4	2	4	3	3
$g_i$	<del>3</del>	<del>2</del>	<del>5</del>	<del>6</del>	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
										

# 核心代码

## ■ 原算法( $O(n^2)$ ):

```
□ for(int i = 0; i < n; i++) {  
    f[i] = 1;  
    for(int j = 0; j < i; j++)  
        if(a[j] < a[i])  
            f[i] = max(f[i], f[j] + 1);  
}
```

## ■ 改进后的算法( $O(n \log n)$ ):

```
□ fill(g, g + n, infinity);  
for(int i = 0; i < n; i++) {  
    int j = lower_bound(g, g + n, a[i]) - g;  
    f[i] = j + 1;  
    g[j] = a[i];  
}
```



## 3.4 容器

# 与容器有关的知识

- 容器(container)是以一定的形式存储一组相同类型的数据的对象。
- STL中的容器都是用模板类来实现的。
- STL中的容器提供了几乎相同的接口。

# 容器的成员函数

- 迭代器操作

- `begin()`, `end()`, `rbegin()`, `rend()`

- 元素操作

- `front()`, `back()`, `operator []()`

- 列表操作

- `insert()`, `erase()`, `clear()`

# 容器的成员函数

- 栈和队列操作

- `push_back()`, `pop_back()`, `push_front()`,  
`pop_front()`

- 其它操作

- `size()`, `empty()`, `resize()`

- 关联容器操作

- `find()`, `lower_bound()`, `upper_bound()`,  
`equal_range()`

# 常用STL容器

名称	描述	所在头文件	迭代器类型
vector	向量	<vector>	随机迭代器
deque	双头队列	<deque>	随机迭代器
list	链表	<list>	双向迭代器
stack	栈	<stack>	不提供迭代器
queue	队列	<queue>	不提供迭代器
priority_queue	优先队列	<queue>	不提供迭代器
set	集合	<set>	双向迭代器
multiset	多重集合	<set>	双向迭代器
map	映射	<map>	双向迭代器
multimap	多重映射	<map>	双向迭代器

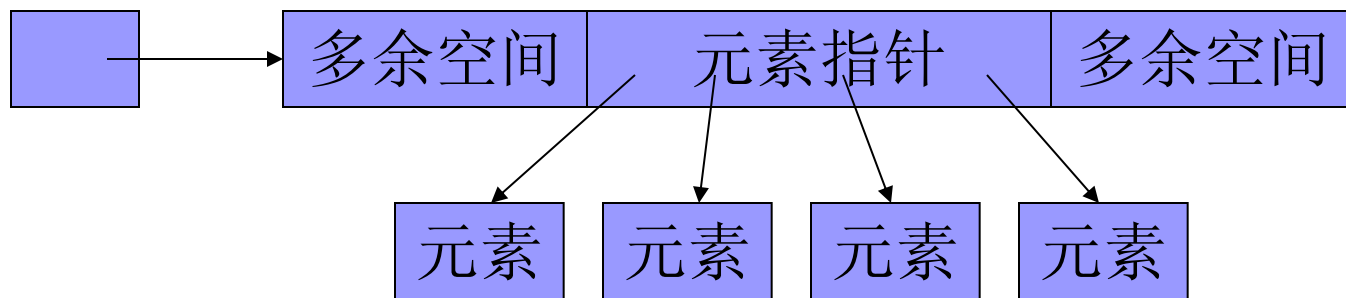


# 一般容器

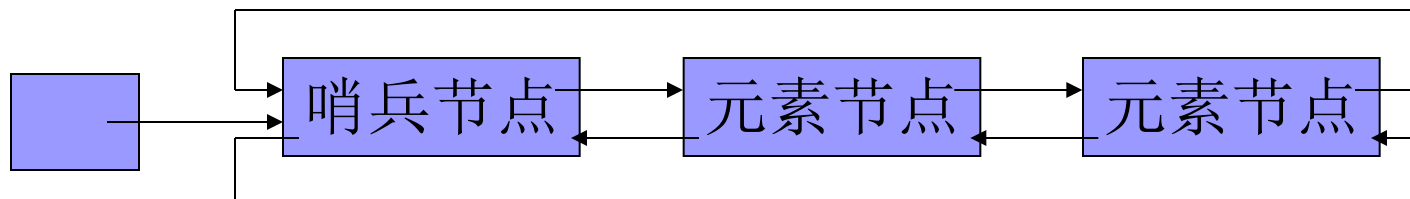
vector



deque



list



# 容器适配器

- 栈stack
- 队列queue
- 优先队列priority\_queue

# 关联容器

- 元素有序
- 用平衡二叉树实现
- 类名： **set, multiset, map, multimap**
- 分类
  - 按元素构成来分
    - 集合：元素本身就是关键字，直接参与排序
    - 映射：元素由关键字和被映射的值构成，只有关键字参与排序
  - 按关键字能否重复来分
    - 普通：关键字不能重复
    - 多重：关键字允许重复

# 关联容器的特殊成员函数

## ■ 查找类

- `find()`
- `lower_bound()`
- `upper_bound()`
- `equal_range()`

## ■ 复合类

- `operator []()`

# 例题：支付帐单

## ■ 题目描述

- 比尔最近遇到了一件麻烦事。每天上午，他会收到若干张帐单（也可能一张也没收到），每一张都有一定的面额。下午，他会从目前还没有支付的帐单中选出面额最大和最小的两张，并把它们付清。还没有支付的帐单会被保留到下一天。现在比尔已经知道他每天收到帐单的数量和面额，请你帮他给出支付的顺序。

## ■ 约束条件


- 天数的上限为30,000
- 每天至少有两张可支付的帐单，保证最后一天全部付清

# 思路分析

- 从数据范围来看，帐单的接收和支付要在  $O(\log n)$  或以下完成。
- 思路一：建一个最大堆和一个最小堆，并在相应元素之间建立映射。
- 思路二：建一棵平衡二叉树。
  - 用STL中的多重集合multiset来实现。

# 核心代码

```
■ multiset<int> bills;
■ for(int i = 0; i < n; i++) {
■     cin >> m;
■     for(int j = 0; j < m; j++) {
■         cin >> a;
■         bills.insert(a);
■     }
■     cout << *bills.begin() << ' '
■         << *--bills.end() << endl;
■     bills.erase(bills.begin());
■     bills.erase(--bills.end());
■ }
```



## 3.5 本章小结




# STL的利弊

## ■ 优点：

- 降低编程复杂度
- 提高代码的正确率

## ■ 缺点：

- 编译时会出各种错误
- 给动态调试增加难度

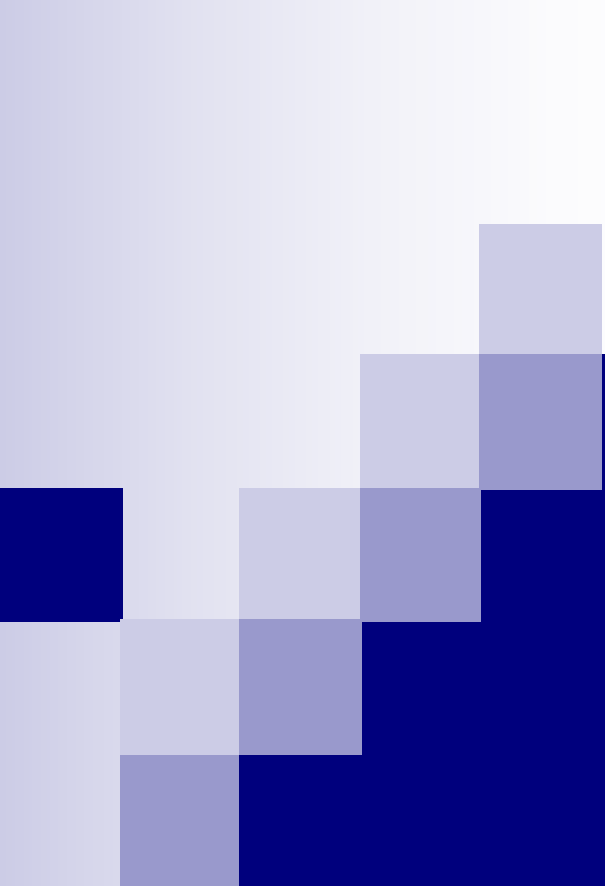


# 几点建议

- 多用**STL**的算法
- 优先使用内置数组
- 多用静态查错
- 动态查错时向屏幕输出

# 总结

- **STL**以面向对象的程序设计和一般化编程为基础，提供了功能强大的算法和容器，并通过迭代器把这两部分有机地结合起来。
- 在信息学竞赛中正确、恰当地使用**STL**可以大大降低编程复杂度，提高代码的正确率，节约宝贵的竞赛时间。
- 但是，**STL**只是一种工具，在竞赛中只能起到辅助的作用。丰富的算法知识、健康的身体素质和良好的心理素质才是竞赛中起决定作用的因素。



# 谢谢大家！

## 欢迎提问！