C++ 标准模板库在 OI 系列比赛的应用

Before Start

OI 中编程环境的改变

- NOI 系列比赛的编程环境，从最早的 Basic，再到之后的 Basic-Pascal，最后到现行的 Pascal-C/C++。
- 在目前 OI 系列比赛学习状况下，Pascal 被广泛认为是较为适合初学者的语言。
- 而在进一步的学习中，更多的人选择投向 C++ 作为自己的语言环境。
- 随着近年来的发展，在 OI 系列比赛中，使用 C++ 的文献和材料也日趋重要和广泛。

C++ 拥有怎样的优势？

- C++ 如同编程语言界的一把瑞士军刀，拥有十分多样的语法和环境。
- C/C++ 的设计理念使得其拥有较高的效率和泛用性。
- C/C++ 在学术上的应用更为广泛。

当然，对于大部分 OI 选手来说，更为重要的是：

C++ 所提供的标准模板库为 OI 系列比赛提供了相当大的便利。

C/C++ 语言语法简述

“我还不会 C++，怎么办？”

没关系，在这一部分中将简要向还没有 C/C++ 语言基础的听众介绍 C/C++ 的基础语法，帮助你理解接下来展示的 C++ 代码。

（不过，完整掌握 C++ 语法还需要更多的练习）

C 的基本语法

- 语句和语句块：
  - 和 Pascal 一样，C 语言的每条语句以分号结尾，无视语句间的空白符号。
  - C 语言的语句块用花括号标识（字符 {} ）相当于 Pascal 的 begin end;
- 定义变量
  - 类型 变量名 1, 变量名 2, ... , 变量名 n;
  - `int a, b, c;`
  - 类型 数组名 [数组大小];
  - `int a[100];`
  - 常量的保留字为 const。
- 条件 `if (A) B;` 循环 `while (A) B; do A while (B);`
  - `for (A; B; C) S` 相当于 `A;while(B){S;C;}`
- 函数定义
  - 返回值 函数名 (参数类型 参数 1, 参数类型 参数 2, ...)
  - `int foobar(int foo, int bar)`

C 的运算符

语法	意义
=	赋值
()、[]、{}	括号/函数调用、数组下标、组合语句块
==、!=	相等、不等
<、>、<=、>=	小于、大于、小于等于、大于等于
+、-、*、/、%	加（正号）、减（负号）、乘、除、取余
!、&&、||	逻辑非、逻辑与、逻辑或
~、&、|、^	二进制取反、二进制与、二进制或、二进制异或
++、--	递增、递减
+=、-= 等运算符 = 的形式	以运算结果赋值
<<、>>	左移、右移
一元 *、一元 &	指针取值、变量取地址

数据类型

- `char`, `short`, `long`, `long long` 分别为 8、16、32、64 位有符号整数类型。
  - `int` 在大部分编译器表现为 32 位整数类型。
  - 加上 `unsigned` 作为前缀后为无符号整数类型。
- `float`, `double`, `long double` 为浮点数类型。
- `struct` 为结构类型，类似于 Pascal 的 `record`
  - 同 pascal 类似，C++ 使用 . 作为结构类型标识符
- `typedef` 定义数据类型的别名。
- 指针变量在变量的名称前加 * 符号。
- 函数的参数为引用传递时加 & 符号。

C++ 的面向对象特性

- 类（`class`）是 C++ 中的自定义数据类型。
  - 与 C 中的 `struct` 相比，类不仅定义了对象所包含的数据，也定义对象所能执行的操作（即成员函数，又称为方法）
    - 许多类都包括构造函数和析构函数这两个成员函数，即创建和删除这个类时调用的函数。
  - C++ 中的 `struct` 被视为是所有成员都被设置为 `public` 的类。
- 类包含两个重要的特性：封装和继承。
  - 类对一部分成员可以限制其访问，隐藏其内部实现，只保留与操作相关的内容。
  - 可以从原有的类中派生出新的类，新的类包含基类的成员。
- 类是面向对象程序设计的重要概念，C++ 是面向对象的编程语言。

C++ 标准模板库的泛用性

- C++ 标准模板库是 C++ 标准库最重要的组成部分。它也是 C++ 的第一个**泛用**的算法/数据结构库。
- 何为泛用？
- 在 C 语言标准库中，所需要的函数与数据类型有关。
  - 如 `printf` 需要使用各种不同的输出格式来标注数据类型，输出格式标注错误则会带来不可预知的结果。
- 而在 C++ 语言的许多标准库中，可以用同样的方法来操作不同的数据结构
  - 如 `iostream` 对于任意数据类型都只需要使用 `>>`，`<<` 来进行输入输出。

容器和算法

程序 = 数据结构 + 算法
——*N.Wirth*

- 容器和算法是 STL 主要包含的两个部分。
- STL 中的容器是一个抽象的数据结构，描述某种泛用地存储数据的方式。
- STL 中的算法旨在提供对于不同容器类型的泛用算法。

泛型算法和函数模板

- 以一个简单的泛型算法为例：<algorithm> 库中的 swap 函数，它的功能是调换两个类型相同的参数的值。
- 如何让这个函数对任何参数都有效？在 G++ 标准库的源码中，这个函数是这么定义的：

```
template<typename _Tp>
inline void
swap(_Tp& __a, _Tp& __b)
```

- 这是一个函数模板，函数模板用于生成泛用的函数
  - template 保留字用于声明模板。
  - 之后的尖括号内声明了模板的参数，其中 typename 保留字表示 _Tp 代表的是参数的类型，这个关键字也可用 class 代替（两者等价）。
  - 在函数的参数中，使用 _Tp 代替参数的类型。

- 在编译过程中，编译器寻找与你编写的代码**最佳契合**的函数模板，然后自动生成对应参数的函数，如同一个函数的生成器。这个展开模板的过程被称为实例化。
  - 有关如何寻找的机制较为复杂，有可能涉及一些奇怪的情况（如模版参数作为函数的返回值时等），在此不做详述。
- 如，在程序中分别使用 `swap` 交换 `int`、`char`、`double`，这些语句会各自链接到三个自动生成的 `swap` 函数。
- 整个过程在编译过程中完成，不占用运行时间，且不会出现运行时的类型错误。

容器和类模板

- 以一个最简单的容器为例：pair，即二元组。它可以包装两个元素，分别作为它的 first 成员和 second 成员。
- pair 内可以包装任何类型的元素，包括其他容器：

```
pair<int, int> PII;
pair<int, double> PID;
pair<pair<int, char>, pair<char, int> > strange_pair;
```

- 定义的这些 pair 变量都是一个类，有自己的构造函数，对应的构造函数使用类似这样。

```
pair<int, double> p2(42, 0.123);
```

- 或者，也可以用 make_pair(A, B) 函数快速地创建 pair。
- pair 的泛用性是如何实现的？显然，这也源于某种类型的模板。

一种可能的 pair 实现的前几行：

```
template<class _T1, class _T2>
struct pair
{
  _T1 first;
  _T2 second;
  const pair()
  : first(), second() { }
  const pair(const _T1& __a, const _T2& __b)
  : first(__a), second(__b) { }
```

- 可以看到，在这个类的定义中，同样使用了 `template` 关键字。
- 与之不同的是，对于在类定义中的模版参数，则是在定义 `pair` 类的时候放在尖括号内给出。
- 在 `pair` 类的成员变量和成员函数，则根据模板的参数来进行实现。
- 因而，可以通过改变 `pair` 容器的参数来制作出存放各种不同数据类型的 `pair` 变量。

运算符重载

min/max 函数也是两个简单的泛型算法。它的可能实现如下。

```
template<class T>
const T& max(const T& a, const T& b)
{
    if (a < b) return b;
    else return a;
}

cout << max(make_pair(1, 2), make_pair(3, 1) ).second
     << endl;
```

代码输出的值是 2，显然，既然 pair 变量可以使用 max 函数，那么它一定支持 pair 之间的比较。

我们发现，定义的两个类型相同的 pair 变量，如果两种子类型都可以用小于符号进行比较，那么这两个 pair 变量也可以直接用小于符号进行比较，这又源于 C++ 中的另一个机制——运算符重载。

```
template<class _T1, class _T2>
const bool operator<(const pair<_T1, _T2>& __x,
 const pair<_T1, _T2>& __y)
{ return __x.first < __y.first
     || (!(__y.first < __x.first)
     && __x.second < __y.second); }
```

观察这段可能的实现代码，这是一段我们之前介绍过的模板函数。所需要注意的是关键字 operator，后面还跟了一个小于符号。

这个函数改变了语言内置的小于符号的运作。对于小于运算符以两个相同的 pair 类作为运算数时，运算的结果将会由这个函数给出。这个过程被称为运算符重载。

在 STL 的使用中常常需要重载一些自定义数据类型之间的运算符，使原有的运算符能够支持自定义数据类型的行为。

当然，这种情况下往往不需要模板，只需要编写对应相关类型的运算符重载函数即可。

```
inline bool operator<(node a, node b){
    return a.v < b.v;
}
```

大部分算术、二进制和逻辑运算符的形式类似 `R operator +(K a, S b);`、比较运算符的形式类似 `bool operator <(K const& a, S const& b);`，累计计算运算符类似 `R& operator +=(K a, S b);`。

对于像 `!~` 这样的一元运算符类似 `R operator !(K a);`，前缀的 `++` 为 `R& operator ++(K a);`，而后缀则为 `R operator ++(K a, int);`

STL 的序列容器

序列容器，即可以被看作是一个序列的容器。

STL 中的序列容器主要有

- `vector`
- `list`
- `deque`

标准库 vector 容器——动态数组

- 动态数组，可作为 C++ 内置数组的替代型态。
- 文件名 `<vector>`
- 原型：

```
template<
class T,
class Allocator = std::allocator<T>
> class vector;
```

- vector 可以直接像数组一样，使用 [] 进行访问。

各种容器大都有的成员函数：clear、size、empty。

vector 容器的大小

- vector 的其中一个构造函数为

```
vector( size_type count,
const T& value = T(),
const Allocator& alloc = Allocator());
```

- 若没有经过构造函数的处理，vector 初始的大小为空，size 方法返回 vector 中元素的数量，而 capacity 方法返回当前存储空间中最多能够存放的元素。
- vector 可以通过 reserve 方法声明来需要更大的存储空间，如果当前存储空间不够，立即重新申请一个那么大的数组并抛弃原来的存储空间。
- vector 可以通过 resize 方法强制改变 vector 的大小，必要时声明更大的存储空间。
- 使用方法 push_back，可以在 vector 尾部插入一个元素。如果这会导致 reserve 或 resize 所保留的的空间不够，立即 resize 到两倍的大小。
- pop_back 弹出最后一个元素，不归还空间。

vector 和数组的差别

- vector 的动态性和封装性好于数组
- vector 的常数有时较大，尤其是滥用 push_back 时。请合理使用手动改变大小/预留空间的 reserve/resize 方法。
- vector 嵌套作为其他容器的参数时更为方便。
- 作为函数的参数时，（`int s[10]`）的 `s` 被视为指针因此值传递时不会复制数组元素。但（`vector<int> s`）的时候 `s` 被视为 vector，因而值传递的时候会复制整个 vector。如果你只需要传递引用，使用 `(vector<int> &s)`。

标准库 list 容器——双向链表

- 双向链表，文件名 <list>
- 原型：

```
template<
    class T,
    class Allocator = std::allocator<T>
> class list;
```

对 list 的访问：STL 的迭代器机制

- 显然，作为一个双向链表，它并不支持随机访问。怎样设计 list 的访问机制更加合适呢。
- 对于数组这样的序列，可以使用指针变量的 ++、−操作遍历数组里面的元素。
- 那么，我们设计一种“智能”的指针，使得任何容器都可以用 ++、−来遍历里面的元素。这种特别的指针被称为迭代器（iterator）。
- STL 兼容的容器的迭代器为其原型类的名称后加`::iterator`，如：`list<int>::iterator`。

- list 有头迭代器和尾迭代器，分别为成员函数的返回值 begin() 和 end()。显然，根据 C/C++ 的管理，头迭代器存放元素而尾指针不存放。
- 遍历 list 元素的方法如下：

```
for (list<int>::iterator it = mylist.begin();
 it != mylist.end(); ++ it)
        std::cout << ' ' << *it;
```

- 迭代器的实现由重载 ++、–、* 三个运算符得到。

迭代器的失效化

- 当然，作为 STL 兼容的容器，vector 也可以用类似的方法进行遍历，但是需要注意一点的是——
- 如果 vector 进行了一次容量扩充。由于其容量扩充是用拷贝所有元素实现，因而之前所有指向 vector 的指针全部会失效。同样，此时所有有关这个 vector 的迭代器都会失去功能，且指向错误的值。
- “失去功能”指可能不能正确使用 ++ 和--进行遍历，这样的问题有时难以排查，需要加以注意。
- 特殊的是，list 的迭代器即使元素被删除也不会失效化，除非两侧的元素也被删除。

list 的成员函数

- `push_back, pop_back, push_front, pop_front`
- `iterator insert( iterator pos, const T& value );`
- `iterator erase( iterator pos );`
  - vector 也有类似的 insert 和 erase 操作，但复杂度不理想。

deque——双端队列

- deque 是一种双端队列，支持快速在两端操作元素，也可以进行随机位置的访问。
- 原型：

```
template<
    class T,
    class Allocator = std::allocator<T>
> class deque;
```

- deque 由块状链表实现。插入操作会无效化所有的迭代器（但不损害其指针特性），删除操作只会影响被操作的节点。

STL 的关联容器

关联容器是实现按照键的查找而非顺序查找的容器。

STL 的关联容器有：

- `set`，`multiset`
- `map`，`multimap`

set——集合

set 定义了一个抽象的集合，可以在集合中查找元素，按大小顺序遍历元素。
set 的使用需要容器内数据类型支持严格小于比较。文件位于 <set>，通常以平衡树（红黑树）实现

原型：

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

set 中不能有重复元素，有重复元素的版本为 multiset，接口完全相同。

函数对象

这个 less 是什么？，显然，这是一个类，我们翻看一下它的定义。

```
template<typename _Tp>
struct less : public binary_function<_Tp, _Tp, bool>
{
  bool
  operator()(const _Tp& __x, const _Tp& __y) const
  { return __x < __y; }
};
```

由此可以看到，less 是一个重载了函数调用运算符 () 的类。对于这种类，我们将其称为函数对象，它是以类型的形式封装的函数行为。

创建的函数对象，如 `less<int> F;`，可以直接像函数一样使用，如 `F(a, b)`。

此外，为了和 C 的兼容性，函数指针也可以被视为函数对象。

set 的成员函数

- set 可以使用迭代器查找大小顺序的上下一个元素。
- `insert erase`
  - 警告：试图删除 end() 会导致不可预知的后果。
- `find count`
- 二分查找：`lower_bound upper_bound`

map——关联映射

map 描述关联的键——值映射关系。文件位于 <map>。

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator =
        std::allocator<std::pair<const Key, T> >
> class map;
```

map 执行 insert 需要的对象是一个 pair，迭代器指向的对象也是 pair。
map 可以使用 [] 运算符，当作“使用 Key 查找 T”的广义数组。

map 的可重复版本为 multimap。

STL 的容器适配器

容器适配器是在已有的容器上实现数据结构的方式。

STL 的容器适配器有：

- `stack`
- `queue`
- `priority_queue`

三种容器适配器

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
template<
    class T,
    class Container = std::deque<T>
> class queue;
template<
    class T,
    class Container = std::vector<T>,
    class Compare =
        std::less<typename Container::value_type>
> class priority_queue;
```

STL 中的算法和其他杂项

algorithm 库中的泛用算法

algorithm 库中包含了若干对于容器的泛用算法，可以用于各种容器。请注意：有的时候对于 algorithm 中的算法无法适应的容器，该容器往往会有同名的成员函数完成类似的功能。

最常见的一些泛用算法如下展示：

sort

```
template <class RandomAccessIterator, class Compare>
  void sort (RandomAccessIterator first,
    RandomAccessIterator last,
    Compare comp);
```

高效的泛用排序算法，针对各种情况都有优秀的发挥，许多人选择 C++ 的理由。

需要随机访问迭代器，可以用于标准数组（普通的数组指针在普通数组中也是随机迭代器）

第三个参数是之前所介绍的函数对象，也可以用 C 风格的函数指针替代。

list 有其成员函数的替代版本。

二分查找

```
template <class ForwardIterator, class T, class Compare>
  ForwardIterator lower_bound (ForwardIterator first,
    ForwardIterator last,
    const T& val, Compare comp);
template <class ForwardIterator, class T, class Compare>
  ForwardIterator upper_bound (ForwardIterator first,
    ForwardIterator last,
    const T& val, Compare comp);
```

找出某一元素出现的最早和最晚位置，只能对已经排序的容器使用。set/map 有其替代版本

去重

```
template <class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique (ForwardIterator first,
  ForwardIterator last, BinaryPredicate pred);
```

去除已排序容器的重复元素，返回去重后的结束位置。

随机打乱

```
template <class RandomAccessIterator, class RandomNumberGen
  void random_shuffle (RandomAccessIterator first,
    RandomAccessIterator last,
    RandomNumberGenerator& gen);
```

bitset

- 封装的二进制压位存储的布尔类型数组，效率很高，适合做压位题。文件 `<bitset>`。
- 模版参数只有一个，即布尔数组的大小。可以像数组一样用 [] 访问。
- bitset 之间重载了二进制运算操作，另有成员函数 count 统计 1 的个数。

STL 的缺憾

STL 的缺陷，大部分来自其抽象性。

- 常数有时还是偏大，需要结合一些正确的使用方法。
- 模板复杂程度高，调试起来十分麻烦，编译错误往往长达十几页。
- 有时会出现一些隐性的错误，如一些不被允许的调用（删除 end）或是操作被无效化的迭代器。
  - 针对这点，可以开启 STL 的调试模式进行一部分的弥补。

G++ 扩展特性

G++ STL 调试模式

- G++ STL 的调试模式是一个针对调试严格编写的 STL 版本，对许多可能的错误都进行了严格的检查。
- 开启 STL 调试模式需要 C++ 标准库目录下存在 debug 目录
- 使用方法：在编译选项中加入`-D_GLIBCXX_DEBUG`，或在程序中加入`#define _GLIBCXX_DEBUG 1`。当检测到问题时，问题将自动输出到 stderr 并中断程序。
- 调试模式会拖慢程序的运行速度，因此在发布时必须将其关闭。

SGI STL——rope

rope 是在 SGI 实现的 STL 版本中加入的新数据类型。string——rope。位置 <ext/rope>，命名空间 __gnu_cxx

rope 是专注于处理大量文本的字符串类型数据结构。rope 可以在 logN 的时间内实现串与串之间的连接和拆分。其接口与 string 大致相同。

范例：

```
#include<ext/rope>
using namespace __gnu_cxx;
int main(){
  crope r(1000000, 'x');
  crope r2 = r + "abc" + r;
  crope r3 = r2.substr(1000000, 3);
  crope r4 = r2.substr(1000000, 1000000);
}
```

TR1——unordered_set 和 unordered_map

Technical Report 1 是 C++03 扩展的一份草稿文件，他涉及了对 C++ 标准库的诸多追加项目。大部分 TR1 的内容都成为了 C++11 的正式组成部分。

`unordered_set` 和 `unordered_map` 是关联容器的哈希表实现，一般状况下拥有更高的效率。位置：`<tr1/unordered_map>`；命名空间：`std::tr1`

原型：

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key,
> class unordered_map;
```

用法和 set 与 map 大同小异，但不存在二分查找或顺序遍历的能力。

TR1——tuple

tuple 是类似 pair 的元组，但它可以存放任意数量的元素。它是基于 TR1 的新特性——可变长度模板构建的。

范例：

```cpp
#include<tr1/tuple>
#include<string>
using namespace std;
using namespace tr1;
int main(){
  tuple<int, string, int> tp = make_tuple(1, "aa", 2);
  get<0>(tp) = 4;
  int x, y; string a;
  tie(x, a, y) = tp; ++ y;
}
```

Policy-Based Data Structures

G++ 的拓展特性，位于 `ext/pb_ds`，命名空间 `__gnu_pbds`。

详情参见《于纪平 _C++ 的 pb_ds 库在 OI 中的应用》