

STL 教程

前置内容

1. 本教程使用 C++11 标准所具有的库和使用方法，C++11 之后的不涉及
2. 查阅 C++ 库的使用方式的网站有如下几个：
 - cppreference.com
 - cplusplus.com/reference/
3. STL 全称为 Standard Template Library, 但是其定义并不是明确的。此教程将会介绍：标准容器，`iostream` 的一部分，算法，智能指针。
4. 由于 STL 是在 `std` 命名空间中的，这里说一下关于 `using` 关键字：使用 `using namespace std` 在许多时候是一种减少重复编程的便利方式，但也有其缺陷。Google 的编程规范明确禁止了 `using namespace xxxxx` 这种用法，因为 "This pollutes the namespace"，一种更好的方式是：

```
#include <iostream>

using std::cin;
using std::cout;
//etc.

// 这样就能够使用 cin 和 cout 了
```

这在大型的项目中是有很大大用处的，具体内容以及更多的规范还可以参考[Google 编程规范](#)（当然在平时的练习中使用 `using namespace std` 也无可非议，这样做有时也是不错的选择）

5. 使用 `nullptr` 而不是 `NULL`
6. STFW (Search The Friendly Web) 浏览互联网以查找某些内容，前几点中已经列出了几个网站可供参考，信息检索对于学习和使用 STL 这个庞大的库是用很大好处的。

标准序列容器

在开始使用序列容器之前，先来看看用 `new` 进行动态分配的恼人之处（引自《Effective STL》）：

1. 必须确保 new 的实例有 delete, 否则就会有资源泄漏;
2. 必须确保正确 delete 或 delete[] (不正确 delete 将会产生未定义行为);
3. 必须确保对一个 new 的实例正确 delete 且只正确 delete 一次

而 C++ 中的容器则可以在适当的时候自动调用析构函数。事实上, 这被称为 RAII (Resource Acquisition Is Initialization) 机制, 是 C++ 一种避免资源泄露的惯用方法。

vector 使用

array 和 vector 都可以与数组做类比, 其中 vector 是动态长度的, 而 array 是静态长度的, 两者相同点在于同一个实例化的容器中的数据类型都是相同的,

```
#include <vector>

using std::vector;

vector<int> v; // 定义一个元素为 int 类型的 vector
vector<int> v[N]; // 定义一个长度为 N 的 vector 数组
vector<int> v(len); // 定义一个长度为 len 的 vector
vector<int> v(len, x); // 定义一个长度为 len, 初始值为 x 的 vector
vector<int> v2(v1); // 拷贝构造
vector<int> v2(v1.begin(), v1.begin() + 3) // 用 v1 前三个元素初始化 v2
```

```

// vector 中的常用内置函数
vector<int> v = { 1, 2, 3 }; // 初始化 vector, v:{1, 2, 3}
vector<int>::iterator it = v.begin(); // 定义 vector 的迭代器, 指向 begin()

v.push_back(4); // 在 vector 的尾部插入元素4, v:{1, 2, 3, 4}
v.pop_back(); // 删除 vector 的最后一个元素, v:{1, 2, 3}

// 注意使用 lower_bound() 与 upper_bound() 函数时 vector 必须是有序的, upper_bound() 在
<algorithm> 中
lower_bound(v.begin(), v.end(), 2); // 返回第一个大于等于 2 的元素的迭代器 v.begin() +
1, 若不存在则返回 v.end()
upper_bound(v.begin(), v.end(), 2); // 返回第一个大于 2 的元素的迭代器 v.begin() + 2, 若
不存在则返回 v.end()

v.size(); // 返回 vector 中元素的个数
v.empty(); // 返回 vector 是否为空, 若为空则返回 true 否则返回 false

v.front(); // 返回 vector 中的第一个元素
v.back(); // 返回 vector 中的最后一个元素
v.begin(); // 返回 vector 第一个元素的迭代器
v.end(); // 返回 vector 最后一个元素后一个位置的迭代器
v.clear(); // 清空 vector
v.erase(iter); // 删除迭代器 iter 所指向的元素
v.insert(iter, 1); // 在迭代器 iter 所指向的位置前插入元素 1, 返回插入元素的迭代器

// 根据下标进行遍历
for (int i = 0; i < v.size(); i++) cout << v[i] << ' ';
// 使用迭代器遍历
for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    cout << *it << ' ';
// foreach 遍历 (C++11)
for (auto x : v)
    cout << x << ' ';

```

判断一个 STL 的对象是否为空应当首选 `empty()` 而不是 `size() == 0`, 这在某些容器中会产生差异巨大的时间差 (参考《Effective STL》)。

注意, 在对 vector 进行操作时要注意分配空间的时间开销。

vector 中有两个不同的概念: `capacity` 和 `size` 前者代表真实占用的内存大小 (某些情况下可以说是预分配的空间), 后者则是容器中所包含的数据大小, 当 `capacity` 不足以容纳 `size` 的时候, vector 会进行扩容, 其计算新空间大小的方式如下:

```
// MSVC 中的实现
_CONSTEXPR20 size_type _Calculate_growth(const size_type _Newsize) const {
    // given _Oldcapacity and _Newsize, calculate geometric growth
    const size_type _Oldcapacity = capacity();
    const auto _Max = max_size();

    if (_Oldcapacity > _Max - _Oldcapacity / 2) {
        return _Max; // geometric growth would overflow
    }

    const size_type _Geometric = _Oldcapacity + _Oldcapacity / 2;

    if (_Geometric < _Newsize) {
        return _Newsize; // geometric growth would be insufficient
    }

    return _Geometric; // geometric growth is sufficient
}
```

可见每次是以 0.5 倍递增的，递增后不够直接使用新的 `size`。

比较以下代码

```
vector<int> v;
vector<int> v(500);
```

在这两种声明之后调用 500 次 `push_back`，第二行代码的内存分配次数小于第一行的内存分配次数，第二行时间开销也小。所以如果能在编程时就知道可能大小，那初始化时指定大小是一种好的办法（但是这种情况使用 `array` 也许会是一种比 `vector` 更好的选择，如何使用可以在学习完这个教程后自行 STFW）

然而想象这样一种情况：程序运行期获取了一个需要的空间大小（或许是 `cin` 的），但编译期并不知道也没有指定（），这时候如果使用 `vector` 该如何尽可能减少内存分配的时间开销？STL 为 `vector` 提供了两个成员函数

- `reserve()`：针对 `capacity`
- `resize()`：针对 `size`（某些情况下也涉及 `capacity`）

来看几个例子

```
// 假设有一个 vector 实例 v, size 为 50, capacity 为 100
v.resize(10);           // size 变为 10, 下标 10 到 49 的元素被删除, capacity 为 100, 没有进行内存分配
v.resize(60);           // size 变为 60, 下标 50 到 59 被填充, capacity 为 100, 没有进行内存分配
v.resize(60, 9999);     // size 变为 60, 下标 50 到 59 被 9999 填充, capacity 为 100, 没有进行内存分配
v.resize(200);          // size 变为 200, capacity 为 200, 重新分配内存
// 忽略前四个
v.reserve(10);          // reserve 未起作用, size 和 capacity 都未发生变化, 元素也没有改变
v.reserve(60);          // 同上
v.reserve(200);         // size 为 50, 元素没有发生变化, capacity 变为 200, 重新分配内存
```

关于 `vector` 内存分配策略的有关内容可以参考[这个](#)

对 `vector` 中的元素进行访问的一个类似于数组的方式就是用 `operator[]`, 也就是类似于 `v[idx]` 这样的方式, 同时 STL 也提供了 `at` 这一成员函数。二者差别在于: `operator[]` 不会检查是否越界, 而 `at` 在越界时会抛出异常, 后续处理可以使用 `try...catch...`, 具体操作可以 STFW 以了解。简单来说就是如果使用 `operator[]`, 那必须要小心下标越界防止程序崩溃, 而 `at` 则可以在越界后进行处理。

string 使用

值得注意的是 `std::string` 是否属于容器是有争议的, 是否是 STL 的一部分也是众说纷纭的。在实际的项目应用中 `std::string` 的使用并不多的, 许多工程项目会选择自己实现一个字符串类型。但是这并不妨碍 `std::string` 成为一个值得学习的内容, 尤其是在快速实现一个想法的时候。而由于 `std::string` 的许多操作与序列容器类似, 故放在此处展开

C++ `string` 类的构造函数有很多, 这里介绍几个可能会用得比较多的, 其它的可以 STFW 自行了解。

```
string(); // 默认
string (const string& str); // 拷贝构造
string (const string& str, size_t pos, size_t len = npos); // 从子串构造
string (const char* s); // 从 C 风格字符串构造
string (const char* s, size_t n); // 从 C 风格字符串前缀构造 (缓冲区)
string (size_t n, char c); // 填充
template <class InputIterator>
    string (InputIterator first, InputIterator last); // 迭代器构造 (类似于 vector)
```

成员函数 (某些功能相近的区别)

```
str.size()    // 字符串长度
str.length()  // 字符串长度（与 size 完全一致）
str.c_str()   // C 风格字符串
```

string 的 insert 和 erase 与 vector 一类容器类似但是有几个更方便的用法：

```
// 以下代码展示 `insert` 的一些用法
#include <iostream>
#include <string>

int main() {
    std::string str = "to be question";
    std::string str2 = "the ";
    std::string str3 = "or not to be";
    std::string::iterator it;

    // used in the same order as described above:
    str.insert(6, str2);           // to be (the )question
    str.insert(6, str3, 3, 4);     // to be (not )the question
    str.insert(10, "that is cool", 8); // to be not (that is )the question
    str.insert(10, "to be ");      // to be not (to be )that is the question
    str.insert(15, 1, ':');        // to be not to be(:) that is the question
    it = str.insert(
        str.begin() + 5, ','); // to be(,) not to be: that is the question
    str.insert(str.end(), 3, '.'); // to be, not to be: that is the question(...)
    str.insert(it + 2, str3.begin(), str3.begin() + 3); // (or )

    std::cout << str << '\n';
    return 0;
}
```

```

// 以下代码展示 erase 用法
#include <iostream>
#include <string>

int main () {
    std::string str ("This is an example sentence.");
    std::cout << str << '\n';

    str.erase(10, 8);
    std::cout << str << '\n';

    str.erase(str.begin() + 9);
    std::cout << str << '\n';

    str.erase(str.begin() + 5, str.end() - 9);
    std::cout << str << '\n';
    return 0;
}

```

// "This is an example sentence."
 // ^
 // "This is an sentence."
 // ^
 // "This is a sentence."
 // ^
 // "This sentence."

除此之外，`string` 还包括了很多与查找有关的成员函数，这里只介绍 `find`

```

// string::find
#include <iostream> // std::cout
#include <string>    // std::string

int main() {
    std::string str("There are two needles in this haystack with needles.");
    std::string str2("needle");

    // different member versions of find in the same order as above:
    std::size_t found = str.find(str2);
    if (found != std::string::npos)
        std::cout << "first 'needle' found at: " << found << '\n';

    found = str.find("needles are small", found + 1, 6);
    if (found != std::string::npos)
        std::cout << "second 'needle' found at: " << found << '\n';

    found = str.find("haystack");
    if (found != std::string::npos)
        std::cout << "'haystack' also found at: " << found << '\n';

    found = str.find('.');
    if (found != std::string::npos)
        std::cout << "Period found at: " << found << '\n';

    // let's replace the first needle:
    str.replace(str.find(str2), str2.length(), "preposition");
    std::cout << str << '\n';

    return 0;
}

```

以上三个示例代码均来自 [C++ Reference](#)

也许会有人问，使用 `vector<char>` 是否是一个好的选择。首先不是所有字符序列都是字符串，有时候使用 `char` 也许只是想存储 **字节**（或 `unsigned char`）此时用 `vector<char>` 会更加清晰。如果是要对字符串有关的内容进行处理，那使用 `vector<char>` 就意味着失去了 `string` 中许多专有的成员函数（当然部分功能也可以通过 `<algorithm>` + 迭代器来实现），所以仍然还是要看具体的使用场景。《Effective STL》中给出了许多 `vector<char>` 可以代替 `string` 的场景，但现在似乎用不到，有兴趣可以查阅。简单来说，当处理对象为 **字符** 的序列时，使用 `string` 是一个不错的选择。

其它

在 `vector` 和 `array` 之外，标准序列容器还有 `array` `deque` `forward_list` `list` 等，有许多操作方式都是类似的，可以 STFW 了解

标准关联容器

对于关联容器这里只介绍 `map` 与 `set`，其余的与序列容器类似可以自己学习。`map` 和 `set` 这一类容器在插入和删除时的效率比其它序列容器高，如果需要频繁插入与删除可以考虑使用 `map` 或 `set`。

`map` 的使用

`map` 提供的是对于一对一式数据的处理，可以自动建立 `key-value` 的对应关系并且可以从 `key` 查找 `value`，其时间复杂度是对数的，对于 1000 个数据，最多查找 10 次，对于 1000000 个数据最多查找 20 次。下面举一个简单的例子来说明 `map` 的用法

```

#include <iostream>
#include <map>
#include <string>
#include <string_view>

void print_map(std::string_view comment, const std::map<std::string, int>& m) {
    std::cout << comment ;

    for (const auto& n : m) {
        std::cout << n.first << " = " << n.second << "; ";
    } // 遍历 map

    // C++ 98 标准中可行的方法
    // for (std::map<std::string, int>::const_iterator it = m.begin(); it != m.end();
it++) {
    //     std::cout << it->first << " = " << it->second << "; ";
    // }
    std::cout << '\n';
}

int main() {
    // 创建一个 map 对象
    std::map<std::string, int> m { {"CPU", 10}, {"GPU", 15}, {"RAM", 20}, };

    print_map("1) Initial map: ", m);

    m["CPU"] = 25; // 更新值
    m["SSD"] = 30; // 插入值
    print_map("2) Updated map: ", m);

    // 对没有的键 (key) 使用 [] 操作符永远都是插入操作
    std::cout << "3) m[UPS] = " << m["UPS"] << '\n';
    print_map("4) Updated map: ", m);

    m.erase("GPU");
    print_map("5) After erase: ", m);

    std::cout << "6) m.size() = " << m.size() << '\n';

    m.clear();
    std::cout << std::boolalpha // 输出 true/false
        << "7) Map is empty: " << m.empty() << '\n';

    return 0;
}

```

set 的使用

头文件: `#include <set>`

set 里的元素是唯一不重复的, 且值不能修改但是能够插入或删除。

初始化方式:

```
std::set<int> myset{1, 2, 3, 4, 5};

int arr[] = {1, 2, 3, 4, 5};
std::set<int> myset(arr, arr + 5); // 使用数组
```

元素的插入

```
std::set<int> myset;
myset.insert(1);
myset.insert(2);
myset.insert(3);
myset.insert(4);
myset.insert(5);
```

set 并不能随机访问, 但是与 map 类似可以使用迭代器或 foreach 风格语句进行遍历, 这里不放代码, 可以仿照 map 和 vector 的遍历方式并 STFW 尝试下

删除指定元素

```
myset.erase(2); // 删除元素 2, `erase` 会返回删除了几个元素, 如果删除了不存在的元素会返回 0, 否则就是 1
```

算法


头文件: `#include <algorithm>`

这里就介绍三个函数, 其它的可以在前面列出的网站中 STFW 自行了解。

使用 `sort` **排序**

```

#include <iostream>
#include <algorithm>
#include <vector>

bool comp(int a, int b) {
    return (a > b); 
}

int main() {

    std::vector<int> vec;
    int n, tmp;

    std::cin >> n;

    for (int i = 0; i < n; ++i) {
        std::cin >> tmp;
        vec.push_back(tmp);
    }
    std::sort(vec.begin(), vec.end()); // 默认
    for (auto &x: vec) {
        std::cout << x << " ";
    }
    std::cout << std::endl;
    std::sort(vec.begin(), vec.end(), comp); // 自定义排序函数
    for (auto &x: vec) {
        std::cout << x << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

lower_bound & upper_bound

这两个函数调用时需要传入两个迭代器位置以及一个值，可选择传入或不传入自定义的比较函数（来自cppreference）：

```
template< class ForwardIt, class T >
ForwardIt lower_bound( ForwardIt first, ForwardIt last, const T& value );

template< class ForwardIt, class T, class Compare >
ForwardIt lower_bound( ForwardIt first, ForwardIt last, const T& value, Compare comp
);

template< class ForwardIt, class T >
ForwardIt upper_bound( ForwardIt first, ForwardIt last, const T& value );

template< class ForwardIt, class T, class Compare >
ForwardIt upper_bound( ForwardIt first, ForwardIt last, const T& value, Compare comp
);
```

其中 `lower_bound` 返回第一个 **大于等于** 给定元素的迭代器，而 `upper_bound` 则是范围第一个 **大于** 给定元素的迭代器

由于 `lower_bound` 和 `upper_bound` 都是二分查找，所以要求传入迭代器的区域是有序的。

拓展阅读

1. [现代 C++ 智能指针](#)