STL 容器

Visual Studio 2015

其他版本

若要了解有关 Visual Studio 2017 RC 的最新文档，请参阅 Visual Studio 2017 RC 文档。

标准库为相关对象的存储集合提供了各种类型安全容器。 容器是类模板；在声明容器变量时，你可以指定该容器将保存的元素类型。 可以使用初始值设定项列表构造容器。 它们具有用于添加和移除元素以及执行其他操作的成员函数。

循环访问的元素在容器中，并通过使用访问单个元素 迭代器。 可以通过使用其成员函数和运算符以及全局函数来显式使用迭代器。 还可以隐式使用它们，例如通过使用范围 for 循环。 所有 STL 容器的迭代器都有一个通用接口，但是每个容器会定义自己的专用迭代器。

容器可以分为三个类别：序列容器、关联容器和容器适配器。

序列容器

序列容器维护你指定的插入元素的顺序。

`vector` 容器的行为类似于数组，但可以根据要求自动增长。 它可以随机访问、连续存储，长度也非常灵活。 基于上述和其他原因，`vector` 是多数应用程序的首选序列容器。 若不确定要使用哪种序列容器，请首先使用矢量！ 有关详细信息，请参阅 vector 类。

`array` 容器具备 `vector` 的某些优点，但长度不够灵活。 有关详细信息，请参阅 array 类。

`deque`（双端队列）容器支持在容器的起点和终点进行快速插入和删除。 它享有 `vector` 随机访问和长度灵活的优点，但是不具备连续性。 有关详细信息，请参阅 deque 类。

`list` 容器是双向链表，在容器内的任意位置启用了双向访问、快速插入和快速删除，但是你不能随机访问此容器中的元素。 有关详细信息，请参阅 list 类。

`forward_list` 容器是单独链表，`list` 的向前访问版本。 有关详细信息，请参阅 forward_list 类。

关联容器

在关联容器中，按照预定义的顺序插入元素，例如按升序排序。 无序的关联容器也可用。 关联容器可分为两个子集：映射和组集。

`map`，有时称为字典，包含键/值对。 键用于对序列排序，值与该键关联。 例如，`map` 可能包含许多键（代表文本中每个独特的单词）和相应的值（代表每个单词在文本中出现的次数）。 `map` 的无序版本是 `unordered_map`。 有关详细信息，请参阅 map 类 和 unordered_map 类。

`set` 仅是按升序排列每个元素的容器，值也是键。 `set` 的无序版本是 `unordered_set`。 有关详细信息，请参阅 set 类 和 _ s e t 类。

`map` 和 `set` 都仅允许将键或元素的一个实例插入容器中。 如果需要元素的多个实例，请使用 `multimap` 或 `multiset`。 无序版本是 `unordered_multimap` 和 `unordered_multiset`。 有关详细信息，请参阅 多重映射类, , unordered_multimap 类, , 多重集合的类, , 和 unordered_multiset 类。

有序的映射和组集支持双向迭代器，其未排序副本支持向前迭代器。 有关详细信息，请参阅 迭代器。

关联容器中的异类查找 (C++14)

排序关联容器（映射、多重映射、集与多重集）现在支持异类查找，这意味着，你将不再需要将完全相同的对象类型作为键或元素在成员函数（如 `find()` 和 `lower_bound()`）中传递。 相反，可以传递为定义了重载 `operator<` 以启用对键类型的比较的类型。

当你在声明容器变量时指定 `std::less<>` 或 `std::greater<>` “菱形函子”比较运算符，会选择性加入启用异类查询：

```
std::set<BigObject, std::less<>> myNewSet;
```

如果你使用默认的比较运算符，该容器的行为会与在 C++11 和更早版本中完全一样。

下列示例演示如何重载 `operator<`，才能使 `std::set` 的用户能够只需通过传入一个可以与每个对象的 `BigObject::id` 成员进行比较的小字符串，来执行查找操作。

```
#include <set>
#include <string>
#include <iostream>
#include <functional>

using namespace std;

class BigObject
{
public:
    string id;
    explicit BigObject(const string& s) : id(s) {}
    bool operator< (const BigObject& other) const
    {
        return this->id < other.id;
```

```
    }

    // Other members....
};

inline bool operator<(const string& otherId, const BigObject& obj)
{
    return otherId < obj.id;
}

inline bool operator<(const BigObject& obj, const string& otherId)
{
    return obj.id < otherId;
}

int main()
{
    // Use C++14 brace-init syntax to invoke BigObject(string).
    // The s suffix invokes string ctor. It is a C++14 user-defined
    // literal defined in <string>
    BigObject b1{ "42F"s };
    BigObject b2{ "52F"s };
    BigObject b3{ "62F"s };
    set<BigObject, less<>> myNewSet; // C++14
    myNewSet.insert(b1);
    myNewSet.insert(b2);
    myNewSet.insert(b3);
    auto it = myNewSet.find(string("62F"));
    if (it != myNewSet.end())
        cout << "myNewSet element = " << it->id << endl;
    else
        cout << "element not found " << endl;

    // Keep console open in debug mode:
    cout << endl << "Press Enter to exit.";
    string s;
    getline(cin, s);
    return 0;
}

//Output: myNewSet element = 62F
```

已重载下列映射、多重映射、集与多重集中的成员函数来支持异类查找：

1. find
2. count

3. lower_bound
4. upper_bound
5. equal_range

容器适配器

容器适配器是序列容器或关联容器的变体，为了简单明确起见，它对接口进行限制。 容器适配器不支持迭代器。

`queue` 容器遵循 FIFO（先进先出）语义。 第一个元素 推送— 即插入队列 — 是第一个要 弹出—，即从队列中删除。 有关详细信息，请参阅 queue 类。

`priority_queue` 容器也是如此组织，因此具有最高值的元素始终排在队列的第一位。 有关详细信息，请参阅 priority_queue 类。

`stack` 容器遵循 LIFO（后进先出）语义。 堆栈上最后推送的元素将第一个弹出。 有关详细信息，请参阅 stack 类。

由于容器适配器不支持迭代器，因此无法与 STL 算法一起使用。 有关详细信息，请参阅 算法。

容器元素的需求

通常，如果插入 STL 容器中的元素可复制，那么这些元素可以是任何对象类型。 只要你不调用尝试复制元素的成员函数，仅可移动的元素（例如，那些类似于 `vector<unique_ptr<T>>` 、使用 `unique_ptr<>` 创建的元素）会一直工作。

析构函数不允许引发异常。

有序的关联容器（本文之前所述）必须已定义公共比较运算符。 （默认情况下，该运算符是 `operator<` ，即使不能与 `operator<` 共同使用的类型也会受支持。）

容器中的某些操作可能还需要公共默认构造函数和公共等效运算符。 例如，未排序的关联容器需要支持相等性和哈希处理。

正在访问容器元素

使用迭代器访问容器的元素。 有关详细信息，请参阅 迭代器。

说明

您还可以使用 基于范围的 for 循环 来循环访问 STL 集合。

比较容器

所有容器都重载运算符 == 用于比较同一类型的两个具有相同的元素类型的容器。 您可以使用 = = 来比较向量 < 字符串>到另一个向量 < 字符串>，，但不能用它来比较向量 < 字符串> 到列表 < 字符串> 或矢量 < 字符串> 对某个向量 < char * >。 您可以使用在 C + + 98/03 std::equal 或 std::mismatch 来比较不同的容器类型和/或元素类型。 在 C + + 11 还可以使用 std::is_permutation。 但在这些情况下函数假设容器都具有相同的长度。 如果第二个范围比第一个短，则产生未定义的行为。 如果第二个范围的更长，结果可能仍然不正确，因为第一个范围结束后比较不会继续。

比较不同的容器 (C++14)

在 C + + 14 及更高版本，您可以通过使用一种比较不同的容器和/或不同的元素类型 std::equal, ，std::mismatch, ，或std::is_permutation 函数采用两个完整范围的重载。 这些重载使你能够比较具有不同长度的容器。 这些重载使用户非常不易遭受错误，并进行了优化，当比较不同长度的容器时会在固定时间内返回错误。 因此，我们建议除非 （1），可以清楚地了解原因无关，或者 （2） 您将使用这些重载 std::list 容器，不会从双范围优化中获得。

来源： https://msdn.microsoft.com/zh-cn/library/1fe2x6kt.aspx