Programming in Modern C++

Module M45: C++ Standard Library (STL): Part 3

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Learnt Standard Template Library (STL) with common components
- Learnt useful containers and their use

Module Objectives

- Summarize containers in STL
- To take a look at a few important library components

Module Outline

Data Structures / Containers in C++

Source:

- Container Classes, isocpp
- Containers library, cppreference
- Standard C++ Library reference: Containers, cplusplus

Data Structures / Containers in C++

- Like `Stack`, several other data structures are available in C++ standard library
- They are *ready-made* and *work like a data type*
- *Varied types of elements* can be used for C++ data structures
- **Data Structures** in C++ are commonly called **Containers**:
  - A container is a *holder object* that stores a *collection of other objects* (its elements)
  - They are implemented as *class templates* allowing great flexibility in the types supported as elements
  - The container
    - *manages the storage space* for its elements
    - provides member *functions to access* them
    - supports *iterators* - reference objects with similar properties to pointers
  - Many containers have several *member functions in common*, and *share functionalities* - easy to learn and remember

Data Structures / Containers in C++

- **Data Structures** in C++ are commonly called **Containers**:
  - `vector`, `list` and `deque` are **Sequence Containers**
  - `map`, `set`, `multimap`, and `multiset` are **Associative Containers**. Also, `unordered_map` (hash table) and `unordered_set` in C++11
  - array (language feature), `string`, `stack`, `queue`, `priority_queue`, and `bitset` are **almost Containers**. C++11 has `array` too
  - `stack`, `queue` and `priority_queue` are implemented as **Container Adaptors**
    - Container adaptors are *not full container classes*, but classes that provide a *specific interface relying on an object of one of the container classes* (such as `deque` or `list`) to handle the elements
    - The underlying container is encapsulated in such a way that its elements are accessed by the members of the container adaptor independently of the underlying container class used

Data Structures / Containers in C++

Container	Class Template	Remarks
Sequence containers: *Elements are* ordered in a strict sequence *and are* accessed by their position in the sequence		
`array` (C++11)	Array class	1D array of *fixed-size*
`vector`	Vector	1D array of *fixed-size* that can *change in size*
`deque`	Double ended queue	*Dynamically sized*, can be expanded / contracted on *both ends*
`forward_list` (C++11)	Forward list	*Const. time insert / erase* anywhere, done as *singly-linked lists*
`list`	List	*Const. time insert / erase* anywhere, iteration in *both directions*
Container adaptors: *Sequence containers* adapted with specific protocols of access *like LIFO, FIFO, Priority*		
`stack`	LIFO stack	Underlying container is `deque` (default) or as specified
`queue`	FIFO queue	Underlying container is `deque` (default) or as specified
`priority_queue`	Priority queue	Underlying container is `vector` (default) or as specified
Associative containers: *Elements are* referenced by their key *and* not by their absolute position *in the container* *They are typically implemented as* binary search trees *and needs the elements to be* comparable		
`set`	Set	Stores *unique elements* in *a specific order*
`multiset`	Multiple-key set	Stores elements in *an order* with *multiple equivalent values*
`map`	Map	Stores *<key, value>* in *an order* with *unique keys*
`multimap`	Multiple-key map	Stores *<key, value>* in *an order* with *multiple equivalent values*
Unordered associative containers: *Elements are* referenced by their key *and* not by their absolute position *in the container* *Implemented using a* hash table of keys *and has* fast retrieval *of elements based on keys*		
`unordered_set` (C++11)	Unordered Set	Stores *unique elements* in no particular order
`unordered_multiset` (C++11)	Unordered Multiset	Stores elements in *no order* with *multiple equivalent values*
`unordered_map` (C++11)	Unordered Map	Stores *<key, value>* in *no order* with *unique keys*
`unordered_multimap` (C++11)	Unordered Multimap	Stores *<key, value>* in *no order* with *multiple equivalent values*

STL Containers

```
template < class T, class Alloc = allocator<T> > class vector; // generic template
template < class T, class Alloc = allocator<T> > class deque;
template < class T, class Alloc = allocator<T> > class list;
template < class T,                    // set::key_type/value_type
    class Compare = less<T>,           // set::key_compare/value_compare
    class Alloc = allocator<T>         // set::allocator_type
    > class set;
template < class T,                    // multiset::key_type/value_type
    class Compare = less<T>,           // multiset::key_compare/value_compare
    class Alloc = allocator<T> >       // multiset::allocator_type
    > class multiset;
template < class Key,                                 // map::key_type
    class T,                                          // map::mapped_type
    class Compare = less<Key>,                        // map::key_compare
    class Alloc = allocator<pair<const Key,T> >       // map::allocator_type
    > class map;
template < class Key,                                 // multimap::key_type
    class T,                                          // multimap::mapped_type
    class Compare = less<Key>,                        // multimap::key_compare
    class Alloc = allocator<pair<const Key,T> >       // multimap::allocator_type
    > class multimap;
template <class T, class Container = deque<T> > class stack;
template <class T, class Container = deque<T> > class queue;
template <class T, class Container = vector<T>,
    class Compare = less<typename Container::value_type> > class priority_queue;
```

STL Containers

Member Type	Definition	Notes
value_type	Template parameter `T`	
allocator_type	Template parameter `Alloc`	defaults to: `allocator<value_type>`
reference	`allocator_type::reference`	for the default allocator: `value_type&`
const_reference	`allocator_type::const_reference`	for the default allocator: `const value_type&`
pointer	`allocator_type::pointer`	for the default allocator: `value_type*`
const_pointer	`allocator_type::const_pointer`	for the default allocator: `const value_type*`
iterator	a random access iterator to `value_type`	convertible to `const_iterator`
const_iterator	a random access iterator to `const value_type`	
reverse_iterator	`reverse_iterator<iterator>`	
const_reverse_iterator	`reverse_iterator<const_iterator>`	
difference_type	a signed integral type, identical to: `iterator_traits<iterator>::difference_type`	usually the same as `ptrdiff_t`
size_type	an unsigned integral type that can represent any non-negative value of `difference_type`	usually the same as `size_t`
key_type	Template parameter `T`	
value_type	Template parameter `T`	
key_compare	Template parameter `Compare`	defaults to: `less<key_type>`
value_compare	Template parameter `Compare`	defaults to: `less<value_type>`

Operations in STL Containers

Container	Capacity	Access	Modifier	Observers	Operations
vector	resize capacity reserve	operator[] at front back	assign push_back pop_back		
deque	resize	operator[] at front back	assign push_back push_front pop_back pop_front		
list		front back	assign push_back push_front pop_back pop_front resize		splice remove remove_if unique merge sort reverse
set				key_comp value_comp	find count lower_bound upper_bound equal_range
multiset				-do-	-do-
map		operator[]		-do-	-do-
multimap				-do-	-do-

Common: (constructor) (destructor) operator= **Iterators**: begin end & rbegin rend **Capacity**: empty size max_size **Modifier**: insert erase swap clear **Allocator**: get_allocator

stack	empty size top push pop
queue	empty size front back push pop
priority_queue	empty size top push pop

algorithm **Component**

Source:

- Algorithms library, cppreference
- <algorithm>, cplusplus

algorithm Component

- The header `<algorithm>` defines a collection of functions especially designed to be used on ranges of elements
- It provides STL-style algorithms
  - Takes one or more sequences
    - Usually as pairs of iterators
  - Takes one or more operations
    - Usually as function objects
    - Ordinary functions also work
  - Usually reports "failure" by returning the end of a sequence

algorithm: Useful algorithms

`r = find(b,e,v)`	`r` points to the first occurrence of `v` in `[b,e)`
`r = find_if(b,e,p)`	`r` points to the first element `x` in `[b,e)` for which `p(x)`
`x = count(b,e,v)`	`x` is the number of occurrences of `v` in `[b,e)`
`x = count_if(b,e,p)`	`x` is the number of elements in `[b,e)` for which `p(x)`
`sort(b,e)`	sort `[b,e)` using `<`
`sort(b,e,p)`	sort `[b,e)` using `p`
`copy(b,e,b2)`	copy `[b,e)` to `[b2,b2+(e-b))` there had better be enough space after `b2`
`unique_copy(b,e,b2)`	copy `[b,e)` to `[b2,b2+(e-b))` but do not copy adjacent duplicates
`merge(b,e,b2,e2,r)`	merge two sorted sequence `[b2,e2)` and `[b,e)` into `[r,r+(e-b)+(e2-b2))`
`r = equal_range(b,e,v)`	`r` is the subsequence of `[b,e)` with the value `v` (basically a binary search for `v`)
`equal(b,e,b2)`	do all elements of `[b,e)` and `[b2,b2+(e-b))` compare equal?

// `b=> .begin()` (start iterator), `e=> .end()` (end iterator), `v=> value`, `p=> p(x)` (predicate)

copy

The copy is available in <algorithm> and it

- Copies the elements in the range [first,last) into the range beginning at result
- Returns an iterator to the end of the destination range (which points to the element following the last element copied)
- The ranges shall not overlap in such a way that result points to an element in the range [first,last)

```
template<class InputIterator, class OutputIterator>
    OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result) {
    while (first!=last) {
        *result = *first; // *res++ = *first++;
        ++result; ++first;
    }
    return result;
}
```

copy: Copy from list to vector

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm> // copy
using namespace std;

void f(vector<double>& vd, list<int>& li) {
        if (vd.size() < li.size()) { cerr << "target container too small" << endl; return; }

        cout << "dst before copy: "; for(auto& x : vd) cout << x << ' '; cout << endl;

        // note: different container types and different element types
        copy(li.begin(), li.end(), vd.begin());
        cout << "dst after  copy: "; for(auto& x : vd) cout << x << ' '; cout << endl;

        sort(vd.begin(), vd.end());
        cout << "dst after  sort: "; for(auto& x : vd) cout << x << ' '; cout << endl;
}
int main() {
    list<int> li = { 2, 7, 5, 6, 8, 9 }; // source container
    vector<double> vd(li.size());        // destination container

    cout << "src before copy: "; for(auto& x : li) cout << x << ' '; cout << endl;
    f(vd, li);
}
```

numeric Component

Source:

- Numerics library, cppreference
- <numeric>, cplusplus

numeric Component

- The header <numeric> defines a set of algorithms (as function templates) to perform certain operations on sequences of numeric values
- Due to their flexibility, they can also be adapted for other kinds of sequences
- The component contains the following algorithms

accumulate	Accumulate values in range
adjacent_difference	Compute adjacent difference of range
inner_product	Compute cumulative inner product of range
partial_sum	Compute partial sums of range
iota [C++11]	Store increasing sequence

accumulate

The accumulate is available in <numeric> and it

- Returns the result of accumulating all the values in the range [first,last) to init
- Uses add as default operation, but a different operation can be specified as binary_op (BinOp)

```
// default
template<class In, class T> T accumulate(In first, In last, T init) {
    while (first!=last) {
        init = init + *first; // init accumulates the result
        ++first;
    }
    return init;
}

// we do not need to use only +, we can use any binary operation (for example, *)
// any function that "updates the init value" can be used:
template<class In, class T, class BinOp> T accumulate(In first, In last, T init, BinOp op) {
    while (first!=last) {
        init = op(init, *first); // means "init op *first"
        ++first;
    }
    return init;
}
```

accumulate: Sum the elements of a sequence

```
#include <iostream>
#include <vector>
#include <numeric> // accumulate
using namespace std;

void f(vector<double>& vd, int* p, int n) {
    double sum = accumulate(vd.begin(), vd.end(), 0.0); // 12.3 : add the elements of vd
    // note: the type of the 3rd argument, the initializer, determines the precision used

    int si = accumulate(p, p+n, 0); // 10 : sum the ints in an int. p+n means (roughly) &p[n]

    long sl = accumulate(p, p+n, long(0)); // 10 : sum the ints in a long

    double s2 = accumulate(p, p+n, 0.0);   // 10 : sum the ints in a double

    // popular idiom, use the variable you want the result in as the initializer:
    double ss = 0;
    ss = accumulate(vd.begin(), vd.end(), ss);  // 12.3 : do remember the assignment
}
int main() { vector<int> v = { 1, 2, 3, 4 };
    int sum = accumulate(v.begin(), v.end(), 0);        // 10

    vector<double> vd = { 1.5, 2.7, 3.2, 4.9 };
    f(vd, &v[0], v.size());
}
```

accumulate: Multiply the elements of a sequence

```
// often, we need multiplication rather than addition:
#include <iostream>
#include <list>
#include <numeric>    // accumulate
#include <functional> // multiplies
using namespace std;

void f(list<int>& ld) {
    int product = accumulate(ld.begin(), ld.end(),
        1.0, // initializer 1.0
        multiplies<int>()); // multiplies is an STL function object for multiplying

    cout << product << endl; // 24
}

int main() {
    list<int> l = { 1, 2, 3, 4 };

    f(l);
}
```

accumulate: What if the data is part of a record?

```
struct Record {
    int units;                  // number of units sold
    double unit_price;
    // ...
};

// let the "update the init value" function extract data from a Record element:
double price(double v, const Record& r) {
    return v + r.unit_price * r.units;
}

void f(const vector<Record>& vr) {
    double total = accumulate(vr.begin(), vr.end(), 0.0, price);
    // ...
}

void f(const vector<Record>& vr) {
    double total = accumulate(vr.begin(), vr.end(), 0.0,   // use a lambda [C++11]
        [](double v, const Record& r) { return v + r.unit_price * r.units; }
    );
    // ...
}
// Is this clearer or less clear than the price() function?
```

inner_product

The inner_product is available in <numeric> and it

- Computes cumulative inner product of range
- Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2
- Uses two default operations (to add up the result of multiplying the pairs) that may be overridden by the arguments binary_op1 (BinOp) and binary_op2 (BinOp2)

```
template<class In, class In2, class T> T inner_product(In first, In last, In2 first2, T init) {
    // This is the way we multiply two vectors (yielding a scalar)
    while(first != last) {
        init  = init + (*first) * (*first2);   // multiply pairs of elements and sum
        ++first; ++first2;
    }
    return init;
}
// we can supply our own operations for combining element values with "init":
template<class In, class In2, class T, class BinOp, class BinOp2 >
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2) {
    while(first!=last) {
        init  = op(init, op2(*first, *first2)); // In default op = operator+ and op2 = operator*
        ++first; ++first2;
    }
    return init;
}
```

inner_product


```
#include <iostream>
#include <vector>
#include <numeric> // inner_product
using namespace std;

int main() {
    // calculate the Dow-Jones industrial index:

    // share price for each company
    vector<double> dow_price = { 81.86, 34.69, 54.45 };

    // weight in index for each company
    vector<double> dow_weight = { 5.8549, 2.4808, 3.8940 };

    // multiply (price, weight) pairs and add
    double dj_index = inner_product(
        dow_price.begin(), dow_price.end(), dow_weight.begin(), 0.0);

    cout << dj_index << endl; // 777.369
}
```

functional Component

Source:

- Standard library header <functional>, cppreference
- <functional>, cplusplus

functional Component

- The header `<functional>` defines a set of useful function objects
- These are typically used as arguments to functions, such as *predicates* or *comparison functions* passed to standard algorithms
- Some useful standard function objects are:
  - Binary
    - `plus`, `minus`, `multiplies`, `divides`, `modulus`
    - `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal`, `less_equal`
    - `logical_and`, `logical_or`
  - Unary
    - `negate`, `logical_not`
- C++11 has heavy use to function objects where the following will be disucssed:
  - `function`, `bind`, `cref`, `ref`, `mem_fn`, ...

Module Summary

- Summarized containers in STL
- Glimpsed at `algorithm`, `numeric`, and `functional` library components