Part 6 — The Standard Template Library (STL)

Overview

STL contains container template classes:

- Sequence containers: elements have predefined locations
  - `vector, slist, list, deque, string` ...
- Associative containers: element location depends on key
  - `set, map, unordered_set, unordered_map` ...

Algorithms:

- Container independent\newline
- `sort, find, merge, random_shuffle` ...

Iterators:

- Pointer-like types used for traversing STL containers
- Interface between algorithms and containers

Example

```cpp
int main() {
    // CONTAINERS + ALGORITHMS
    vector<int> v(10);                      // vector of 10 ints
    generate(v.begin(), v.end(), rand);     // fill with random values
    v[0] = 6;                               // access like array

    // loop through vector using ITERATORS (old school)
    // C++11: global begin/end prefered
    int sum=0;
    vector<int>::iterator it = begin(v), en = end(v);
    for (; it != en; ++it) {
        sum += *it;
    }
    // new in C++11: range-based for
    for (const int &x : v) {
        cout << x << endl;
    }

    // ALGORITHM: shuffle elements randomly
    random_shuffle(begin(v), end(v));

    // MEMBER FUNCTIONS: if non-empty, erase first element
    if (!v.empty()) {
        v.erase(begin(v));
    }
}
```

Overview

STL is part of the C++ standard library. Several implementations exist.

STL websites:

- www.sgi.com/tech/stl
- https://en.cppreference.com/w/
- www.cplusplus.com/reference

Good Books:

- Josuttis: *The C++ Standard Library*
- Meyers: *Effective STL*

Sequence Containers

For sequence containers the user specifies each element's location

E.g., node in linked lists, or index in arrays or vectors

Elements can be inserted and removed by indicating their position

This is in contrast to associative containers which store element based on their keys

vector<T>

- Vector template, dynamic array functionality
- Element type is T
- Sequence that allows random access to elements of type T by index
- Simple STL container, often most efficient one
- Vectors can grow and shrink
- Amortized constant time insertion/deletion at the end
- Linear time insertion/removal anywhere else
- Iterators or pointers that refer to vector elements are invalidated by insert/delete operations when the underlying array is re-allocated
- Switch index check on with

```
g++ -D_GLIBCXX_DEBUG ...
```

Example

```cpp
#include <vector>
using namespace std;

int main() {
    const int N = 1000;
    vector<int> v;          // empty integer vector
    v.reserve(N);           // reserve memory for N elements. saves time and memory because v
                            // because v doesn't need to grow; v.size() still 0

    // append N elements
    for (int i=0; i < N; ++i) { v.push_back(i); }

    // add up all elements, array syntax
    int s = 0;
    for (size_t i=0; i < v.size(); ++i) { sum += v[i]; }
    // alternative: use iterator to step through vector (C++98):
    // a bit faster because above v.size() is called multiple times
    s = 0;
    vector<int>::iterator it = begin(v), en = end(v);
    for (; it != en; it++) { s += *it; }

    for (const auto &x : v) { s += x; }             // or in C++11 simply:

    // remove all elements one by one back to front
    while (!v.empty()) { v.pop_back(); }
    assert(v.empty());      // When leaving main v is destroyed here
                            // However, if v contains pointers,
                            // destructors are *NOT* called on the objects the pointers point to
                            // They have to be destroyed in a loop first!
}
```

Frequently Used Vector Methods

```
// iterator, reference, size_type are vector-local types
// defining the vector iterator, element reference, and
// index types. They can be accessed via vector<T>::...

iterator begin() : returns iterator to first element
iterator end()   : returns iterator to end (pointing behind last element)
                   (C++11: global begin/end preferred)

size_type size()   : # of elements in vector
bool empty() const : true iff vector is empty (may be faster than !size())

void push_back(const T &):  inserts new element at the end
                            (amortized constant time)
void pop_back()         :   destroys last element, decreases size,
                            doesn't shrink capacity

reference operator[](size_type i): returns reference to element i
reference back()   : returns reference to last element (assumes size() > 0)

void clear()              : remove all elements
void erase(iterator pos)  : removes element at position pos
void reserve(size_type n) : sets capacity to at least n (size() unchanged)
bool operator==(const vector &, const vector &) : element-wise equality
```

`list<T>`

- `#include <list>`
- `list<T>` is a doubly linked list
- Data type associated with nodes is \ve{T}
- Allows forward/backward traversal
- If backward traversal is not needed, use `slist<T>` (singly linked list)
- Constant time for insertion/removal of elements anywhere (at known location)
- Inserting/deleting elements does not invalidate iterators

Example

```cpp
#include <list>
#include <iostream>

int main() {
    list<int> l;
    l.push_back(0);
    l.push_front(1);
    l.insert(begin(l), 2);          // same as l.push_front(2), l now 2 1 0

    std::list<int> x(3, 10);        // x is list of 3 tens
    l.splice(++begin(l), x);        // x now empty

    // new C++11 feature: auto - infers rhs type
    // instead of list<int>::iterator it = ...
    auto it = begin(l), en = end(l);

    for (; it != en; ++it) {
        std::cout << *it << " ";
    }
    // output: 2 10 10 10 1 0

    // even shorter: C++11's range-based for
    // (const reference x steps through container)
    for (const auto &x : l) {
        std::cout << x << " ";
    }
}
```

Frequently Used List Methods

```
iterator begin()            : returns iterator to first element
iterator end()              : returns iterator to end (last+1)
                              (C++11: global begin/end prefered)
size_type size()            : # of list elements (linear time!)
bool empty() const          : true iff list is empty

void push_front(const T &)  : inserts new element at the front
void push_back(const T &)   : inserts new element at the end
void pop_front()            : removes first element
void pop_back()             : removes last element

iterator insert(iterator pos, const T &) : inserts element in front of pos
void erase(iterator pos) : removes element at position pos

void reverse()   : reverses list (linear time!)
void splice(iterator pos, list<T> &x) : inserts x in front of pos, clears x
```

Other Sequence Containers

`deque<T>`

- `#include <deque>`
- Double-ended queue; supports random access: `d[i]`
- Inserting/deleting at both ends takes amortized constant time
- Inserting/deleting in the middle: linear time

`basic_string<T>`

- `#include <string>`
- Sequence of characters

  `std::string = basic_string<char>}`

- Similar to `vector`
- Many member functions:

  `insert, append, erase, find, replace,` …

Example

```cpp
#include <string>
std::string s("foo");           // constructed from C-string
std::string t("bar");

std::cout << s + t;             // concatenation : foobar
if (s < t) ...                  // lexicographic ordering

if (s.empty()) ...              // check if empty

std::cout << s.size();          // number of characters (3)

s.push_back('x');               // foox

std::cout << s[3];              // access by index (x)

std::cin >> s;                  // read white-space separated word

getline(std::cin, s);           // read next line into s

// replace first occurrence of foo by bar
s.replace(s.find("foo"), 3, "bar");

// get access to raw C-string
const char *cstr = s.c_str();
```

Associative Containers

- Support efficient retrieval of elements based on keys
- Support insertion/removal of elements
- Difference to sequences: no mechanism for inserting elements at specific locations — element locations are fully determined by keys
- Element keys cannot be modified. I.e., `*it = x` is not allowed for iterator `it` pointing to an element of a associative container
- But associated data can be modified via iterators. E.g.

```
1 std::map<int,double>::iterator it = ...
2 it->second = 3;          // now element is mapped to 3
```

Set

`set<T>`

- `#include <set>`
- Simple unique associative container
- Keys are the elements themselves
- No two elements are the same
- Internally, sets are represented as balanced binary search trees (e.g., red-black trees)
- Elements are stored in nodes
- Logarithmic time for find / insert / delete elements\newline (I.e., with n elements stored, the worst-case time for above operations is $\Theta(\log n)$)
- Inserting/deleting elements does not invalidate iterators (except for those pointing to deleted elements)
- Traversing sets will visit all elements in sorted order (low $\longrightarrow$ high)

Example

```cpp
#include <set>
std::set<int> s;
s.insert(0); s.insert(2);
s.insert(1); s.insert(0);       // populate s, s = { 0, 1, 2 }

// enumerating all elements stored in set (C++98)
std::set<int>::iterator it = begin(s), en = end(s);
for (; it != en; ++it) {
    std::cout << *it << ' ';
}
// equivalent C++11 code:
for (auto it=begin(s); it != end(s); ++it) {
    std::cout << *it << ' ';
}
// also equivalent:
for (const auto &x : s) {
    std::cout << x << ' ';
}

if (s.find(0) != end(s)) {
    std::cout << "found element";
}
// output: 0 1 2 found element
```

Comparison Functors for Associative Containers

Building search trees for sets and maps is based on key comparisons. This is implemented with the help of functors.

Functors are classes that define `operator()`. Invocations look like function calls:

```
Foo f;
f(a);
```

Sets and maps require functors with two arguments for comparing two elements:

```
struct MyCompare {
    // return true iff a < b
    bool operator()(const T &a, const T &b) {
        ...
    }
};

set<int> s;                 // default: functor less<int> is used,
                            // defined for all built-in types
set<int, MyCompare> s;      // using my own compare functor instead
```

Comparison Functors for Associative Containers

For sets and maps to work, the binary relation $<$ implemented by functors must be a ***strict weak ordering***, i.e. $<$ is a partial ordering:

- Irreflexivity: for all x: $x < x$ false (**important, often missed!**)
- Antisymmetry: for all x, y: if $x < y$ then not $(y < x)$
- Transitivity: for all x, y, z: if $x < y$ and $y < z$ then $x < z$

and equivalence is transitive (x and y are called equivalent iff not $(x \leq y)$ and not $(y \leq x)$).

Total orders are strict weak orderings with ***equivalent = identical***. The usual $<$ for `int` and lexicographic ordering for strings are total orderings and so, set and maps will work with them.

Example — `set` with Functor

```cpp
#include <set>
struct Point { int x, y; };

// compare points in lexigographic order
struct CompPoint {
    // return true iff a < b
    bool operator()(const Point &a, const Point &b) {
        int d = a.x - b.x;
        if (d < 0) { return true; }
        if (d > 0) { return false; }
        return a.y < b.y;
    }
};
std::set<Point, CompPoint> point_set;        // set of Points
```

Somewhere in `set<T,C>` implementation:

```cpp
C f;                    // Our functor as a template type argument
// ...
if (f(a, b)) {          // a < b?
    // ...
}
```

Frequently Used Set Methods

```
iterator begin()    : returns iterator to first element
iterator end()      : returns iterator to end (last+1)

size_type size()    : number of set elements
bool empty() const  : true iff set is empty

pair<iterator, bool> insert(const T &x) :  inserts element; if new, returns
                                           (iterator,true) - otherwise (?,false)

    pair<iterator, bool> p = s.insert(5);
    if (p.second) {
        // We inserted a new element ...
    }

void erase(iterator it)  : removes element pos points to
void clear()             : remove all elements

iterator find(const T &x) const : looks for x, returns its
                                  position if found, and end() otherwise

set_union(), set_intersection(), set_difference() : set ops
```

Map

`map<Key, Data>`

- `#include <map>`
- Sorted-pair-unique associative container
- Associates keys with data
- Value-type is `pair<const Key, Data>`
- Insert/delete operations do not invalidate iterators (except for those pointing to deleted elements)
- Also based on binary search trees
- Like sets, but additional data item associated with key

Example

```cpp
#include <map>

// handy type abbreviation
// Month2Days now refers to std::map<std::string, int>
typedef std::map<std::string, int> Month2Days;
Month2Days m2d;

// the following is equivalent to:
// m2d.insert(std::pair<std::string,int>("january", 31));
m2d["january"]   = 31; m2d["february"] = 28;
m2d["march"]     = 31; m2d["april"]    = 30;
m2d["may"]       = 31; m2d["june"]     = 30;
m2d["july"]      = 31; m2d["august"]   = 31;
m2d["september"] = 30; m2d["october"]  = 31;
m2d["november"]  = 30; m2d["december"] = 31;
// Careful - [] operator creates pair if it doesn't exist in map yet!

string m = "june";
Month2Days::iterator cur = m2d.find(m);
// const auto cur = m2d.find(m);

// pair.first contains key (can't be changed)
// pair.second contains associated date (can be changed)
if (cur != end(m2d)) {      // we found it
    std::cout << m << " has " << cur->second << " days" << std::endl;
} else {
    std::cout << "unknown month: " << m << std::endl;
}
```

Frequently Used Map Methods

```
iterator begin()    : returns iterator to first pair
iterator end()      : returns iterator to end (last+1)

size_type size()    : number of pairs in map
bool empty() const  : true iff map is empty

void clear()             : erase all pairs
void erase(iterator pos) : removes pair at position pos

pair<iterator, bool> insert(const pair<Key,Data> &):
    inserts (key,data) pair, returns iterator and true iff new

iterator find(const Key &k) : looks for key k, returns its position if
                              found, and end() otherwise

Data &operator[](const Key &k) : returns the data associated with key k;
                                 if it does not exists inserts default data value!
```

Iterators

Generalization of pointers.

- Often used to iterate over ranges of objects.
- Iterator points to object
- The incremented iterator points to the next object

Central to generic programming

- Interface between containers and algorithms
- Algorithms take iterators as arguments
- Container only needs to provide a way to access its elements using iterators
- Allows us to write generic algorithms operating on different containers such as vector and list using the same code

Iterator Hierarchy

Input Iterator, Output Iterator

- Only single pass (like reading/writing file) `v = *it;` (no increment/decrement needed)
- Read or write access, respectively — writing to input iterators not supported, nor reading from output iterators

Forward Iterator

- can be used to step through a container several times (read or write)
- only "++" supported (e.g., `std::slist`)

Iterator Hierarchy

Bidirectional Iterator

- Motion in both directions "`++ --`",
- e.g., `std::list`

Random Access Iterator

- in addition allows adding of offsets to iterators
- e.g., `*(it+5)`

Ranges

- Most algorithms are expressed in terms of iterator ranges [begin, end)
- begin points to first element, end points PAST the last element
- Range is empty iff `begin() == end()`
- `[begin(v), end(v))` represents the range of all elements in container v
- `end()` is sometimes used to indicate failure
- E.g. linear search (`find`) returns `end()` to indicate an unsuccessful search
- Dereferencing `end()` is an error

Const Iterators

Sometimes we need access to const containers. For this purpose STL provides const iterators

```
1  struct X {
2      void print() const {
3          // the following does not work, because ...::iterator gives write
4          // access which clashes with print being const
5          //
6          // std::vector<int>::iterator it = begin(data), en = end(data);
7
8          std::vector<int>::const_iterator it = begin(data), en = end(data);
9          // this works because *it is read-only
10         // shorter: auto it = begin(data), ...
11         for (; it != en; ++it) { ... }
12
13         // C++11: auto it = begin(data)... or
14         for (auto &x : data) {
15             // ...
16         }
17     }
18
19     // Member
20     std::vector<int> data;
21 };
```

Reverse Iterators

Iterator adaptor that enables backwards traversal of a range using `operator++`

```cpp
#include <iterator>

vector<int> v;

v.push_back(1); v.push_back(2);

auto rit = v.rbegin();
// type vector<int>::reverse_iterator
// pointing to last element

auto rend = v.rend();
// pointing to element "in front of" first

// traverse v backwards
while (rit != rend) {
    cout << *rit++ << endl;
}

// output: 2 1
```

Non-Mutating STL Algorithms

Non-Mutating algorithms work on ranges but do not change elements

- `for_each` : apply a function to each element
- `find` : find an element
- `equal` : checks whether two ranges are the same
- `count` : count elements equal to value
- `search` : search for a sub-sequence

Example: for_each

```cpp
#include <set>
#include <algorithm>

// functor with state
struct Add {
    int sum;

    Add() { sum = 0; }
    void operator()(int x) { sum += x; }
};

set<int> s;
s.insert(1); s.insert(2); s.insert(3);

Add f = std::for_each(begin(s), end(s), Add());

cout << f.sum << endl;          // 1+2+3 = 6
// how does this work?
```

Example: for_each

```
1 template <class InpIterator, class UnaryFunc>
2 UnaryFunc for_each(InpIterator begin, InpIterator end, UnaryFunc f) {
3     for (; begin != end; ++begin) {
4         f(*begin);
5     }
6     return f;
7 }
```

Applies function or functor f to each element in [begin, end)

Returns the function object after it has been applied to all elements in [begin, end)

Mutating STL Algorithms

Work on range and possibly change elements

- `remove_if` : moves elements for which a predicate is false to front, returns `new_end`, size unchanged
- `partition` : reorders `elements;` `x` with `pred(x)=true` come first
- `generate` : assigns results of function calls to elements
- `copy` : copies input range to output iterator
- `fill` : assigns a value to each element
- `reverse` : reverses range
- `rotate` : general rotation of range w.r.t. to mid-point
- `random_shuffle` : randomly shuffles all elements
- `sort` : sorts a range

There are many more ...

Example

```cpp
#include <algorithm>

// functor
struct Even {
    bool operator()(int x) { return !(x & 1); }
};

const int N = 20;
vector<int> v, w;
int A[N];

partition(begin(v), end(v), Even());    // even | odd
generate(begin(v), end(v), rand);

copy(begin(v), end(v), begin(w)); // dangerous! w must be large enough
copy(begin(v), end(v), back_inserter(w));
// better : translates *it = v; into w.push_back(v)

fill(begin(v), end(v), 314159);
reverse(A, A+N);    // array viewed as STL container; works!
rotate(begin(v), begin(v)+1, end(v));   // "<<< 1"

random_shuffle(A, A+N);
```

Sort

```
1 // version 1, uses operator less<T>
2 template <typename RandomAccessIterator>
3 void sort(RandomAccessIterator first, RandomAccessIterator end);
4
5 // version 2 , uses comparison functor less
6 template <typename RandomAccessIterator, typename StrictWeakOrdering>
7 void sort(RandomAccessIterator first, RandomAccessIterator end, StrictWeakOrdering less);
```

- Sorts random access range in non-decreasing order
- Implements "*introspection sort*" which combines quicksort and heapsort
- Worst and average case time complexity: $\Theta(n \log n)$
- **FAST!** Template can inline comparison function and Introspection Sort is faster than Quicksort in worst-case
- Renders C-library's qsort obsolete

Example

```cpp
#include <algorithm>
#include <functional>   // for less<T>, greater<T> ...

std::vector<int> v(10);
const int N = 20;
int A[N];

// fill range with random values, works for functions and functors!
std::generate(std::begin(v), std::end(v), rand);

// works for arrays, too!
std::generate(A, A+N, rand);

// non-decreasing order, uses less<int>
std::sort(std::begin(v), std::end(v));

// non-increasing
std::sort(std::begin(v), std::end(v), std::greater<int>());

// also works for arrays! non-decreasing
std::sort(A, A+N);

// non-increasing
std::sort(A, A+N, std::greater<int>());
```

What else is there in STL?

- Hashed associative containers (called `unordered_set/map}`)
  - E.g., `unordered_set<T,HashFunc,EqualKey>` (now incorporated into C++11)
  - Organized as hash table
  - Faster than the standard tree-based containers — $O(1)$ access, rather than $\Theta(\log n)$
  - But needs more space
  - See www.cplusplus.com/reference/unordered_set/unordered_set
- More sorting related functions (`stable_sort, merge, lower_bound`)
- Global `begin/end` functions (which are more general than member functions and now preferred).

```
vector<int> v; auto it = begin(v) ...
```