STL Containers

Container basics

An STL container is a collection of objects of the same type (the elements).

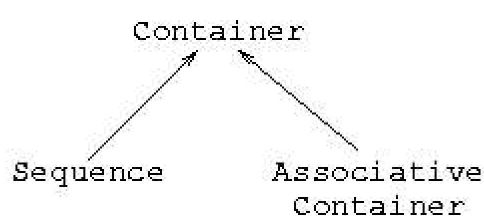
- Container *owns* the elements.
  - Creation and destruction is controlled by the container.

Two basic types of containers:

- Sequences
  - User controls the order of elements.
  - `vector`, `list`, `deque`
- Associative containers
  - The container controls the position of elements within it.
  - Elements can be accessed using a *key*.
  - `set`, `multiset`, `map`, `multimap`

Container concepts

There are three main container concepts:



Container concepts are not as important for generic programming as iterator concepts.

- There are fewer models.
- Containers have important properties that are not described by the basic container concepts.
  - Properties that differentiate one container from another.
  - In contrast, an iterator is almost fully described by the most refined concept it models.
  - More refined container concepts would have just one model.
- Even the basic concepts can be too refined:
  - `boost::array`
- There are almost no generic algorithms taking a container as an argument.
  - insert iterators

However, container concepts standardize basic features.

- Consistent interface makes using them easier.
- Fairly easy to replace one container with another.

The `Container` concept

Properties shared by all STL containers.

- default constructor
- copy constructor and assignment
    - deep copy
- `swap`
    - `a.swap(b)` and `swap(a, b)`
    - constant time
- `==`, `!=`
    - content-based equality: equal elements in same order
- order comparisons
    - lexicographic order: first inequal elements determine the order

    ```
    vector<int> a, b;
    // a = [1, 2, 3]
    // b = [1, 3, 2]
    assert(a < b);
    ```

- `begin()`, `end()`
- `size()`, `empty()`, `max_size()`
- member types
    - `value_type`
    - `reference` (to the value type)
    - `const_reference`
    - `iterator`
    - `const_iterator`
    - `difference_type` (as with iterators)
    - `size_type` (often unsigned type, usually `size_t`)

In addition, a *reversible* container has the properties:

- `rbegin()`, `rend()`
- member types
    - `reverse_iterator`
    - `const_reverse_iterator`

Sequences

Common properties of all sequence containers:

- constructors
    - Fill constructor `Container(n, val)` fills container with `n` copies of `val`.
    - Default fill constructor `Container(n)` fills container with `n` default constructed values.

  - Range constructor `Container(i, j)` fills container with the contents of the iterator range `[i,j)`.
- `assign`
  - fill assignment `assign(n, val)`
  - range assignment `assign(i, j)`
  - old elements are assigned to or destroyed
- `insert`
  - `insert(p, val)` inserts `val` just before the position pointed by iterator `p`.
  - `insert(p, n, val)` inserts `n` copies.
  - `insert(p, i, j)` inserts the contents of range `[i,j)`.
- `erase`
  - `erase(p)` erases the element pointed by iterator `p`.
  - `erase(p,q)` erases the range `[p,q)`
  - returns iterator to the position immediately following the erased element(s)
- `clear()` erases all

vector

`vector` should be used by default as the (sequence) container:

- It is more (space and time) efficient than other STL containers.
- It is more convenient and safer than primitive array.
  - automatic memory management
  - rich interface

Properties of `vector` in addition to sequences:

- `v[i]`, `at(i)`
  - `v.at(i)` checks that `0 <= i < v.size()`
- `front()`, `back()`
  - return reference to the first and last element (not beyond last)
- `push_back(val)`
  - inserts `val` to the end
- `pop_back()` removes
  - removes the last element and returns it
- `resize`
  - change the number of elements in the vector
  - `resize(n)` makes `n` the size; fills with default values if necessary
  - `resize(n, val)` fills with `val` if necessary
- `capacity()`, `reserve(n)` (see below)

Memory management

- The elements are stored into a contiguous memory area on the heap.
  - `capacity()` is the number of elements that fit into the area.
  - `size()` is the actual number of elements. The remainder of the area is unused (raw memory).
  - `reserve(n)` increases the capacity to `n` without changing the size.
  - The capacity is increased automatically if needed due to insertions.
- Capacity increase may cause copying of all elements.

- A larger memory area is obtained and elements are copied there.
- Capacity increase by an insertion doubles the capacity to achieve *amortized constant time*.

- Capacity never decreases.
    - Memory is not released.
    - But the following gets rid of all extra capacity/memory:

```
vector<int> v;
...
vector<int>(v).swap(v);  // copy and swap
```

- Use `&v[0]` to obtain a pointer to the memory area.
    - May be needed as an argument to non-STL functions.

```
vector<char> v(12);
strcpy(&v[0], "hello world");
```

Limitations of vector

- Insertions and deletions in the beginning or in the middle are slow.
    - Requires moving other elements.
    - Prefer `push_back()` and `pop_back()`.
    - Insert or erase many elements at a time by using the range forms of `insert` and `erase`.
- Insertions and deletions *invalidate* all iterators, and pointers and references to the elements.

```
vector<int> v;
...
vector<int> b = v.begin();
v. push_back(x);
find(b, v.end());  // error: b is invalid
```

deque

`deque` stands for double-ended queue. It is much like `vector`.

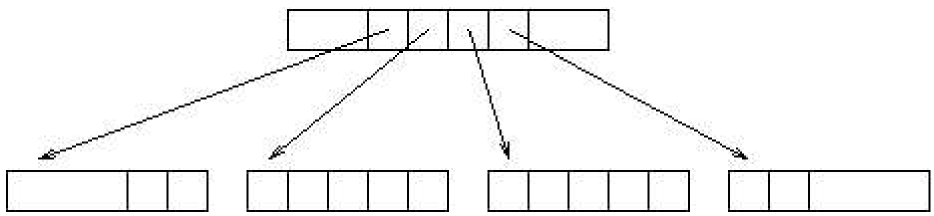
- Differences to `vector`
    - Insertion and deletion in the beginning in (amortized) constant time.
        - `push_front`, `pop_front`
    - Slower element access and iterators.
    - No `capacity()` or `reserve(n)` but also less need for them.
    - Insertions and deletions to the beginning and end do not invalidate pointers and references to other elements.
        - But iterators may be invalidated.

```
deque<int> d(5,1);
deque<int>::iterator i = d.begin() + 2;
```

```
int* p = &*i;
d.push_front(2);
int x = *p;  // OK
int y = *i;  // error: i may be invalid
```

Memory management

`deque` stores the elements something like this:



- Element access and iterators are more complicated.
- Fast insertions and deletions to the beginning.
- Handles size changes gracefully: - Capacity increases or decreases one block at a time.
- Memory area is not contiguous.

`list`

The third standard sequence container is `list`. The underlying data structure is a doubly-linked list:

- No random access.
- Fast insertion and deletion anywhere.
- Insertions and deletions do not invalidate iterators, pointers or references to other elements.

Member functions in addition to sequence:

- `push_front`, `pop_front`
- `push_back`, `pop_back`
- `splice`
  - `c1.splice(i1, c2)` removes all elements from list `c2` and inserts them at position `i1` in list `c1`.
  - `c1.splice(i1, c2, i2)` removes the element pointed by `i2` from `c2` and inserts it at position `i1` in list `c1`.
  - `c1.splice(i1, c2, i2, j2)` removes the range `[i2,j2)` from `c2` and inserts it at position `i1` in list `c1`.
  - In the last two cases, `c1` and `c2` can be the same list.
  - constant time
  - Iterators, pointers and references keep pointing to the same element even if it is moved to a different list.

```
template <class T, class A>  // A is allocator
void catenate (list<T,A>& c1, list<T,A>& c2) {
  c1.splice(c1.end(), c2);
}
```

- member versions of STL algorithms
  - `reverse`, `sort`, `merge`, `remove`, `remove_if`, `unique`

- In some cases, like `sort`, the algorithm would not work for lists because it requires random access iterators.
- With the member versions, iterators, pointers and references keep pointing to the same element, unless that element is deleted.

```
list<int> c;
c.push_back(10);
c.push_back(20);
c.push_back(30);
list<int>::iterator i = c.begin();
assert(*i==10); // i points to the first element: 10
c.reverse();
assert(*i==10); // i continues to point to 10
                // which is now to the last element
reverse(c.begin(), c.end());
assert(*i==30); // i still points to the last element
                // which now contains 30
```

string

`string` class was designed before STL, but STL container properties were added to it later. It is similar to vectors; the differences include:

- only `char` as element type
- many additional operations
  - concatenation (`operator+`, `append`)
  - I/O (`<<`, `>>`, `getline`)
  - C-string conversions
  - `substr`, `compare`, `find`, `replace`
  - Many operations can take C-string or substring as an argument.
  - Many of the operations could be replaced with STL algorithms.
- Many implementations have optimizations:
  - reference counting with COW (copy on write)
  - short string optimization

vector<bool>

`vector<bool>` has some special properties.

- Elements are stored as bits in a bit vector.
  - very space-efficient
  - some operations may be slow
- It is not possible to have a pointer or a reference to a bit.
  - `operator[]`, `front()`, `back()`, and iterator's `operator*` do not return a reference but a *proxy* object that behaves almost like a reference but not quite.
  - Taking address of a proxy and assigning it to a reference is not possible.

```
vector<bool> v;
// ...
bool* p = &v[0];    // illegal
bool& r = v.back(); // illegal
```

- Otherwise a proxy can be used on either side of an assignment.

```
bool tmp = v[0];
v[0] = v[1];
v[1] = tmp;
```

- Does not satisfy all container requirements and iterators do not satisfy all requirements of random access iterators.

- `flip`
  - `v.flip()` flips all bits
  - flip one bit: `v[1].flip()`, `v.back().flip()`, `v.begin()->flip()`

Associative containers

The STL standard associative containers (`set`, `multiset`, `map`, `multimap`) allow access to elements using a *key*:

- For `set` and `multiset` element is its own key.
- For `map` and `multimap` elements are of type `pair<const Key, T>`.
  - `pair` is a standard template class defined as:

```
template <class T, class U>
struct pair {
  T first;
  U second;

  // some constructors
};
```

- `set` and `map` contain at most one element for each key.
- `multiset` and `multimap` can contain many elements with the same key.

The underlying data structure is a balanced search tree:

- logarithmic access time
- requires order comparisons of keys
- iteration in key order
- Iterators, pointers and references stay valid until the pointed to element is removed.

The order comparison

- `operator<` by default but can be changed

```
struct my_less {
  bool operator() (int a, int b) { return a < b; }
}

// all three sets have the same ordering:
set<int> s1;  // default: operator< as key order
set<int, std::less<int> > s2;
set<int, my_less> s3;
```

- Two keys are *equivalent* if neither is smaller than the other.
  - `operator==` is not used for comparing keys.
  - Ensures consistency of order and equivalence.

- must be *strict weak ordering*:
  1. *irreflexivity*: `x<x` is always false.
  2. *transitivity*: `(x<y) && (y<z)` implies `x<z`.
  3. *transitivity of equivalence*: if `x` equals `y` and `y` equals `z`, then `x` equals `z`.

     ```
     // NOT strict weak ordering:
     // 1 equals 2 and 2 equals 3 but 1 does not equal 3
     struct clearly_less {
       bool operator() (int a, int b) { return a < b-1; }
     }
     ```

  - asymmetry: `x<y` implies `!(y<x)`.
    - Often mentioned as a requirement, but it is implied by 1. and 2.
    - Often called antisymmetry in STL literature, but asymmetry is the correct mathematical term.

Common properties

In addition to properties of the `Container` concept, all associative containers have:

- member types
  - `key_type`
  - `key_compare`
- comparison operators
  - `key_comp()` returns the key comparison operator
  - `value_comp()` returns a comparison operator comparing elements not keys.
- constructors
  - Range constructor `Container(i,j)` fills container with the contents of the range `[i,j)`.
  - (All constructors accept a comparison object as an extra optional argument.)
- `insert`
  - `insert(x)`
  - `insert(i, x)`. Iterator `i` is a hint pointing to where the search for insertion position should start.
    - Allows `insert_iterator` to operate on associative containers.
  - range insert `insert(i, j)`
  - For `set` and `map` insertions are not done if the key is already there.
- `erase`
  - `erase(k)` erases all elements with key `k`
  - `erase(i)` erase element pointed to by `i`
  - range erase `erase(i,j)`
- searching
  - `find(k)` returns iterator to the element with key `k` or `end()` if no such element
  - `count(k)`
  - `lower_bound(k)` find first element with key not less than `k`
  - `upper_bound(k)` find first element with key greater than `k`
  - `equal_range(k)` returns `pair<iterator,iterator>` representing the range of element with key `k`

set and multiset

- Defined in header file `set`.
- Implement the abstract data structures of set and multiset.
- There are no additional member operations.

```
// count distinct words
set<string> words;
string s;
while (cin >> s) words.insert(s);
cout << words.size() << " distinct words\n";
```

There are no member operations for set intersection, union, etc. However, the following generic algorithms work on any sorted range, including `[s.begin(),s.end())` for a set or multiset `s`:

- `includes`
- `set_intersection`
- `set_union`
- `set_difference`
- `set_symmetric_difference`

```
string str1("abcabcbac");
string str2("abcdabcd");

multiset<char> mset1(str1.begin(), str1.end());
multiset<char> mset2(str2.begin(), str2.end());

multiset<char> result;
set_intersection (mset1.begin(), mset1.end(),
                  mset2.begin(), mset2.end(),
                  inserter(result, result.begin()) );

copy(result.begin(), result.end(), ostream_iterator<char>(cout));
// outputs: "aabbcc"
```

map and multimap

- Defined in header file `map`.
- `multimap` has no additional operations, `map` has one, `operator[]`.
- The elements are pairs, which can make insertion and access slightly awkward.
  - `operator[]` is the most convenient way:

    ```
    map<string, int> days;
    days["january"] = 31;
    days["february"] = 28;
    // ...
    days["december"] = 31;
    if (leap_year) ++days["february"];

    cout << "February has " << days["february"] << " days.\n";
    ```

  - `multimap` does not have `operator[]`. The helper function `make_pair` is useful:

```
multimap<string, string> children;
children.insert(make_pair("Jane","Ann"));
children.insert(make_pair("Jane","Bob"));
children.insert(make_pair("Bob","Xavier"));
// ...

typedef multimap<string, string>::iterator iterator;
pair<iterator,iterator> answer;
answer = children.equal_range("Jane");
cout << "Jane's children:";
for (iterator i = answer.first; i != answer.second; ++i)
  cout << " " << i->second;
```

Container adaptors

The container adaptors `stack`, `queue`, `priority_queue` are containers implemented on top of another container.

They provide a limited set of container operations:

- member types `value_type` and `size_type`, `container_type`
- basic constructors, destructors and assignment
- contruction from the underlying container `adaptor(const container&)`
- comparison operators
- `size()`, `empty()`

Stack

`stack` can be implemented on top of `vector`, `deque` or `list`.

- The default is `deque`.

```
// these are equivalent
stack<int> st1;
stack<int, deque<int> > st2;
```

Additional operations:

- constructor `stack(const container&)`
- `push(val)`
- `top()`
- `pop()`

Queue

`queue` can be implemented on top of `deque` or `list`.

- The default is `deque`.

Additional operations:

- `front()`
- `back()`
- `push(val)`

- `pop()`

Priority queue

`priority_queue` can be implemented on top of `deque` or `vector`.

- The default is `vector`.

Uses order comparison operators of elements similar to the associative containers.

```
// these are equivalent
priority_queue<int> pq1;
priority_queue<int, vector<int> > pq2;
priority_queue<int, vector<int>, less<int> > pq3;
```

Additional operations:

- range constructor
- comparison object as an extra optional argument of constructors
- `push(val)`
- `top()` returns the largest element
- `pop()` removes the largest element

There is no method for changing the priority of an element or removing an element that is not the largest.

- Sufficient for some applications like event simulation.
- Not sufficient for others like Dijkstra's algorithm.

Hash tables

The standard has no containers using hashing, but they are a common extension. They are also included in the Technical Report on C++ Standard Library Extensions, commonly known as TR1, an extension of the standard library likely to be included in the next C++ standard:

- `unordered_set`
- `unordered_multiset`
- `unordered_map`
- `unordered_multimap`

More details can be found in the proposal.

Container elements

The type of container elements should always be a model of the `Assignable` concept:

- Normally behaving copy constructor and copy assignment.
- Never store into a container types that are not assignable:
  - references (no assignment)
  - `std::auto_ptr` (abnormal copy behavior)

Some member functions have additional requirements:

- default constructor
  - default fill constructor `Container(n)`
- equality comparisons
  - containers's `operator==`
- order comparisons
  - container's `operator<`
  - associative container with default order comparison

Pointers in containers

Pointers as container elements require special care. There two kinds of pointers:

- Pointers that do not own the object they point to.
  - Example: the same element in multiple containers.
    - for example, different iteration orders
    - One container stores the elements, others store pointers to the elements.
  - Prefer iterators to pointers.
    - They enable container manipulation.
  - Be careful about validity
    - With `deque` use pointers instead of iterators if there are insertions or deletions at the beginning or the end.
    - With `vector` use index if there are insertions or deletions at the end.

- Pointers that own the element they point to.
  - Example: polymorphic container:

```
struct animal {
  virtual ~animal() {};
  virtual void eat() =0;
  // ...
}

struct baboon : animal {
  // ...
}

struct lion : animal {
  // ...
}

// ...

vector<animal*> zoo;
zoo.push_back(new baboon);
zoo.push_back(new lion);
zoo[1]->eat();
```

  - Such pointers are problematic elements.
    - Containers take care that the destructor of an element is called, when the element is erased (or the container is destroyed).
    - But an owning pointer's destructor does not do what it should: destroy the pointed to object and release the memory.
    - The user of the container must ensure that the pointed to objects are properly destroyed and freed. This is inconvenient and error-prone.

        - Achieving exception safety is difficult.

    - Better to use a *smart pointer*, whose destructor does the right thing.

        - `auto_ptr` has the right kind of destructor, but unfortunately the wrong kind of copy constructor and assignment.
        - Use `boost::shared_ptr` if possible.

```
typedef boost::shared_ptr<animal> animal_ptr;
vector<animal_ptr> zoo;

animal_ptr p(new baboon);
zoo.push_back(p);
p.reset(new lion);
zoo.push_back(p);
zoo[1]->eat();
```

Exception safety

Exceptions are a common error reporting mechanism in C++. The elements of STL containers are allowed to throw exceptions (except in destructors). In particular, if a copy of an element fails and throws an exception during a container operation, one of the following guarantees are provided:

- *Strong guarantee*: The container operation is cancelled, and the container's state is as if the operation was never called.
    - Most list operations.
    - All single element operations on lists and associative containers.
    - push and pop operations on vector and deque.
- *Basic guarantee*: There are no memory leaks, and the container is in a consistent state. In particular, the container can be destroyed safely.

These guarantees make it possible to recover from an exception without memory leaks.