C++ Standard Template Library

Overview STL contains six major kinds of components: *containers*, *generic algorithms*, *iterators*, *function objects*, *adaptors*, and *allocators*.

Containers There are two categories of STL container types: *sequence containers* and *sorted associative containers*.

Sequence containers organize a collection of objects, all of the same type `T`, into a strictly linear arrangement. Examples are:

- `vector<T>`
- `deque<T>`
- `list<T>`

Sorted Associative containers provide an ability for fast retrieval of objects from the collection based on keys. There are four sorted associative container types:

- `set<Key>`
- `multiset<Key>`
- `map<Key, T>`
- `multimap<Key, T>`

Iterators STL generic algorithms are written in terms of iterator parameters, and STL containers provide iterators that can be plugged into the algorithms. **Iterators** are pointer-like objects that STL algorithms use to traverse the sequence of objects stored in a container. There are five iterator categories: *random access*, *bidirectional*, *forward*, *input* and *output iterators*. See Figure 1 to see which type of operations are supported by the iterators.

Input iterators are used to read values from a sequence. An example is its use in `find` algorithm as shown below:

```
template <typename InputIter, typename T>
InputIter find(InputIter first, InputIter last,
               const T& value) {
  while (first != last && *first != value)
    ++first;
  return first;
}

int main() {
  // find the first elt equal to 7 in array
  int a[5] = {12, 3, 25, -7, 8};
  int *ptr = find(&a[0], &a[5], 7);

  list<int> lst(&a[0], &a[10]);
  list<int>::iterator i = find(lst.begin(),
                               lst.end(), 7);

  // print the first char after 'x'
  istream_iterator<char> in(std::cin);
  istream_iterator<char> std::eos;
  find(std::in, eos, 'x');
  std::cout << *(++in) << std::endl;
}
```

Output iterators allow us to write values into a sequence. An example is its use in **copy** algorithm, which copies from one sequence to another:

```
template <typename InIter, typename OutIter>
OutIter copy(InIter first, InIter last,
             OutIter result) {
  while (first != last) {
    *result = *first;
    ++first;
    ++result;
  }
}
```

A **forward iterator** is one that is both an input iterator and an output iterator, and it thus allows both reading and writing and traversal in one direction.

```
template <typename ForwardIter, typename T>
void replace(ForwardIter first,
             ForwardIter last,
             Const T& x, const T& y) {
  while (first != last) {
    if (*first = x) {
      *first = y;
    ++first;
  }
}
```

A **bidirectional iterator** is similar to a forward iterator, except that it allows traversal in either direction. STL **reverse** algorithm uses bidirectional iterators.

While above iterators only allow sequence accesses to elements in containers, a **random access iterator** allows access to any position inside a sequence in constant time. For example, STL generic **binary_search** algorithm uses random access iterators.

Following is a meaning of some common iterator functions:

- **front**(): first element
- **begin**(): pointer to the first element
- **back**(): last element
- **end**(): pointer to last-plus-one element position

Generic Algorithms Many generic algorithms have a version which accepts a *function parameter*. For example, sorting algorithms accept a binary predicate parameter which compares two values.

Nonmutating sequence algorithms are those which does not modify the containers which operate. Examples include: `find`, `adjacent_find`, `count`, `for_each`, `mismatch`, `equal`, and `search`.

Mutating sequence algorithms modify the contents of the containers on which they operate. For example, the **unique** algorithm eliminates all consecutive duplicate elements from a sequence. Other algorithms are: `copy`, `fill`, `generate`, `shuffle`, `remove`, `replace`, `reverse`, `swap`, and `transform`.

Sorting-related algorithms are related to sorting. For example, algorithms for sorting and merging sequences and for searching and performing set-like operations on sorted sequences.

operation	input	output	forward	bidir	random
copy/copy-constructor (`X b(a), b = a`)	○	○	○	○	○
increment (`a++, *a++`)	○	○	○	○	○
equality (`==, !=`)	○		○	○	○
dereferenced as rvalue (`*a, a->m`)	○		○	○	○
dereferenced as lvalue (`*a = t, *a++ = t`)		○	○	○	○
default constructor (`X a; X()`)			○	○	○
decrement (`a--; *a--`)				○	○
arithmetic operators (`+ , -`)					○
comparisons (`> , <, >=, <=`)					○
compound assignment (`+= , -=`)					○
offset dereference (`a[n]`)					○

Figure 1: Operations supported by iterators

Function Objects A *function object* is any entity that can be applied to zero or more arguments to obtain a value and/or modify the state of the computation. In C++, any ordinary function satisfies this definition, but so does an *object of any class that overloads the function call operator*, `operator()`.

```
class multiply {
public:
  int operator()(int x, int y) const {
    return x*y;
  }
};
multiply multfuncobj;
int product1 = multfuncobj(3,7);
```

Consider the STL generic function `accumulate`, which adds up `init` plus all the values between `first` and `last`:

```
template <typename InputIter, typename T>
T acculmulate(InputIter first, InputIter last,
              T init) {
  while (first != last) {
    init = init + *first;
    ++first;
  }
  return init;
}
```

If we want to replace the "addition" with a new two-argument function, we can modify the above function template as follows:

```
template <typename InputIter, typename T>
T acculmulate(InputIter first, InputIter last,
              T init, T (*binfunc)(T x, Ty)) {
  while (first != last) {
    init = (*binfunc)(init, *first);
    ++first;
  }
  return init;
}
```

Adaptors Adaptors are STL components which can be used to change the interface of another component. They are defined as template classes that take a component type as a parameter. There are three categories of adapters: *container adaptors*, *iterator adaptors*, and *function adaptors*.

Container adaptors are used to provide a new data structure such as `stack` or `queue`, using existing STL containers such as `vector` or `list`. Examples are: *stack container adaptor*, *queue container adaptor*, and *priority queue adaptor*. Consider a stack container adaptor which can be applied to a `vector`, `list`, or `deque`. The *stack container adaptor*, `stack<T>`, is a stack of `T` with a default implementation using a `deque`. `stack<T, vector<T> >` is a stack of `T` with a `vector` implementation. Also, `stack<T, list<T> >` can be used.

Iterator adaptors are STL components which can be used to change the interface of an iterator component. Only one kind of iterator adaptor is defined in STL: *reverse iterator adaptor*, which transforms a given bidirectional or random access iterator into one which the traversal direction is reversed.

Function adaptors help us construct a wider variety of function objects. Using function adaptors is often easier than directly constructing a new function object type with a struct or class definition. There are three categories of function adaptors: *binders*, *negators*, and *adaptors for pointers to functions*. A **binder** is a function adaptor which converts binary function objects into unary function objects by binding an argument to some particular value.

```
int *where = find_if(&a[0], &a[1000],
             bind2nd(greater<int>(), 200));
```

A **negator** is used to reverse the sense of predicate function objects. There are two negator adaptors: `not1` and `not2`.

```
int *where = find_if(&a[0], &a[1000],
             not1(bind2nd(greater<int>(), 200)));
```

An **adaptor for pointers to functions** is provided to allow pointers to unary and binary functions to work with the other function adaptors provided in the library.

Allocators C++ STL uses special objects, called *allocators* to handle the allocation and deallocation of memory. An allocator defines a *memory model* and it is useful to enforce special memory models, such as such as shared memory, garbage collection, and object-oriented databases, without changing interfaces.

Basic allocator operations are as follows:

- `a.allocate(n)`: allocate memory for *n* elements
- `a.construct(p)`: initialize the element to which *p* refers
- `a.destroy(p)`: destroy the element to which *p* refers
- `a.deallocate(p, n)`: deallocate memory for *n* elements to which *p* refers