**Container and Iterators**

# STL

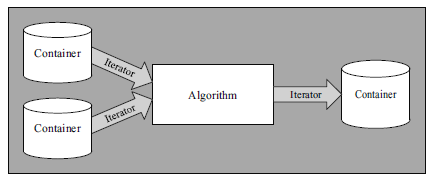
The STL is a generic library that provides solutions to managing collections of data with modern and efficient algorithms. It’s called the Standard Template Library (STL), and was **developed by Alexander Stepanov and Meng Lee of Hewlett Packard**. The STL is part of the Standard C++ class library, and can be used as a standard approach to storing and processing data.

From the programmer’s point of view, the STL provides a bunch of collection classes that meet various needs, together with several algorithms that operate on them.

All components of the STL are templates, so they can be used for arbitrary element types. But the STL does even more: It provides a framework for supplying other collection classes or algorithms for which existing collection classes and algorithms work.

STL has three important components:

1. Container
2. Iterators
3. Algorithms



STL Components

# Container

Containers are used to manage collections of objects of a certain kind. The containers may be implemented as arrays or as linked lists, or they may have a special key for every element. Every kind of container has its own advantages and disadvantages, so having different container types reflects different requirements for collections in programs.

## Sequence containers

Sequence containers are ordered collections in which every element has a certain position. This position depends on the time and place of the insertion, but it is independent of the value of the element.

Sequence containers are usually implemented as arrays or linked lists.

1. array (C++11)
2. vector
3. dequeue
4. list
5. forward\_list (C++11)

## Associative containers

Associative containers are sorted collections in which the position of an element depends on its value (or key, if it’s a key/value pair) due to a certain sorting criterion. If you put six elements into a collection, their value determines their order. The order of insertion doesn’t matter.

Associative containers are usually implemented as binary trees.

1. set
2. mutiset
3. map
4. multumap

## Unordered associative containers

Unordered (associative) containers are unordered collections in which the position of an element doesn’t matter. The only important question is whether a specific element is in such a collection. Neither the order of insertion nor the value of the inserted element has an influence on the position of the element, and the position might change over the lifetime of the container. Thus, if you put six elements into a collection, their order is undefined and might change over time.

Unordered containers are usually implemented as hash tables.

The major advantage of unordered containers is that finding an element with a specific value is even faster than for associative containers.

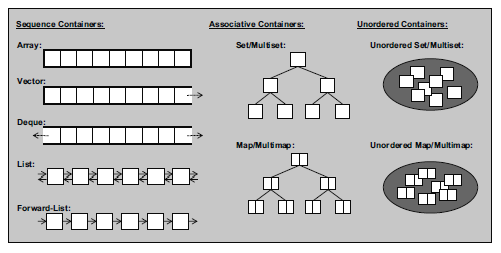
In fact, the use of unordered containers provides amortized constant complexity, provided that you have a good hash function. However, providing a good hash function is not easy, and you might need a lot of memory for the buckets.

1. unordered\_set
2. unordered\_multiset
3. unordered\_map
4. unordered\_multumap

## Container adaptors

Container adapters, which are predefined containers that provide a restricted interface to meet special needs. These container adapters are implemented by using the fundamental container classes.

1. stack
2. queue
3. priority\_queue



STL Container Types

# Algorithms

Algorithms are used to process the elements of collections. For example, algorithms can search, sort, modify, or simply use the elements for various purposes.

Algorithms are not member functions of the container classes but instead are global functions that operate with iterators. This has an important advantage: Instead of each algorithm being implemented for each container type, all are implemented only once for any container type. The algorithm might even operate on elements of different container types. You can also use the algorithms for user defined container types.

# Iterators

We need a concept of an object that represents positions of elements in a container. This concept exists. Objects that fulfil this concept are called iterators.

An iterator is an object that can iterate over elements (navigate from element to element). These elements may be all or a subset of the elements of an STL container. An iterator represents a certain position in a container. The following fundamental operations define the behaviour of an iterator:

* **Operator \*** returns the element of the current position. If the elements have members, you can use operator -> to access those members directly from the iterator.
* **Operator ++** lets the iterator step forward to the next element. Most iterators also allow stepping backward by using operator --
* **Operators == and !=** return whether two iterators represent the same position
* **Operator =** assigns an iterator (the position of the element to which it refers).

These operations provide exactly the interface of ordinary pointers in C and C++ when these pointers are used to iterate over the elements of an ordinary array. The difference is that iterators may be smart pointers — pointers that iterate over more complicated data structures of containers.

The internal behaviour of iterators depends on the data structure over which they iterate. Hence, each container type supplies its own kind of iterator. As a result, **iterators share the same interface but have different types**. This leads directly to the concept of generic programming: Operations use the same interface but different types, so you can use templates to formulate generic operations that work with arbitrary types that satisfy the interface.

In fact, every container defines two iterator types:

1. container::iterator is provided to iterate over elements in read/write mode.
2. container::const\_iterator is provided to iterate over elements in read-only mode.

# END