**Comparison between STL Containers**

There are a few rules of thumb that apply to selecting which container to use:

• Unless you have a reason to use another container, use a vector.

• If your program has lots of small elements and space overhead matters, don’t use list or forward\_list.

• If the program requires random access to elements, use a vector or a deque.

• If the program needs to insert or delete elements in the middle of the container,use a list or forward\_list.

• If the program needs to insert or delete elements at the front and the back, but

not in the middle, use a deque.

**Usage of containers**

**Ayes of Vector**

• If all you want is a "smart array" class that offers random access to elements, and perhaps the ability to append new elements to its end, then vector is the right choice.

• Accessing individual elements by their position index (constant time).

• Iterating over the elements in any order (linear time).

• Add and remove elements from its end (constant amortized time).

**Nays of Vector**

• When you find yourself, frequently inserting, removing elements in the middle of the vector.

• If you think about inserting elements before the first element, then clearly vector is not the right choice because it doesn't even have a push\_front() operation.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Ayes of Deque**

• Deque provides random access and the subscript notation for accessing elements, just like vector. Yet unlike vector, it supports efficient insertions and deletions at both ends.

• Individual elements can be accessed by their position index.

• Iteration over the elements can be performed in any order.

• Elements can be efficiently added and removed from any of its ends (either the beginning or the end of the sequence).

• E.g. Operating system's scheduler.

**Nays of Deque**

• When you find yourself, frequently inserting, removing elements in the middle of the deque.

• Calling insert() in the middle of the deque invalidates all the iterators and references to its elements.

• Calling insert() at either end of the deque invalidates all its iterators but has no effect on the validity of references to its elements. After the push\_front() call, the iterator it becomes invalid whereas the pointer p remains valid.

• Similarly, calling erase() in the middle of the deque invalidates all the iterators and references to its elements.

• Calling erase() at either end of the deque invalidates only the iterators and the references to the erased elements. deque<int> di; int p\* = &di[5]; deque<int>::iterator it=di.begin()+5; di.push\_front(8);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Ayes of List**

• Efficient insertion and removal of elements anywhere in the container (constant time).

• If you don't need random access (say, "15 positions from the beginning" or "three positions before the current element") and you do need to insert new elements between existing elements frequently, then list is a good choice.

• Efficient moving elements and block of elements within the container or even between different containers: splice operation (constant time).

• E.g. Window’s task manager

**Nays of List**

• When is list a bad choice? First and foremost, – When you need random access to elements. – If you need to sort the elements frequently. Although list does offer the sort()operation, sorting lists isn't as efficient as sorting vectors.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Ayes of Map**

• As associative containers, they are especially designed to be efficient accessing its elements by their key (unlike sequence containers, which are more efficient accessing elements by their relative or absolute position).

• Unique key values: no two elements in the map have keys that compare equal to each other

• Each element is composed of a key and a mapped value.

**Ayes of Set**

• Internally, the elements in a set are always sorted from lower to higher following a specific strict weak ordering criterion set on container construction.

• Unique element values: no two elements in the set can compare equal to each other. For a similar associative container allowing for multiple equivalent elements, see multiset.

• The element value is the key itself. For a similar associative container where elements are accessed using a key, but map to a value different than this key, see map.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Summary**

• If all you want is a "smart array" class that offers random access to elements, and perhaps the ability to append new elements to its end, then vector is the right choice.

• When you need to insert or remove elements at either end of a container frequently, but you rarely use insertions and deletions in the middle, deque is a good choice.

• If you don't need random access (say, "15 positions from the beginning" or "three positions before the current element") and you do need to insert new elements between existing elements frequently, then list is a good choice.

• As associative containers, they are especially designed to be efficient accessing its elements by their key (unlike sequence containers, which are more efficient accessing elements by their relative or absolute position).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

 [14.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-14-638.jpg?cb=1421209622" \t "_blank" \o ) Adaptors • Adaptor containers change ordinary containers such as vector and deque, making them look as different container types. • For example, a stack is usually implemented as a list with a different interface. Instead of push\_back() and pop\_back(), it has push() and pop().

 [15.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-15-638.jpg?cb=1421209622" \t "_blank" \o ) Stack • Stack (declared in <stack> ) is an ideal choice when you need to a LIFO data structure. • For example, think about people entering the back seat of a car that has only one door: the last person to enter is the first to exit. • The four basic operations that a stack supports are push(), pop(),top(), and empty().

 [16.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-16-638.jpg?cb=1421209622" \t "_blank" \o ) Queue • A queue (declared in <queue> ), or FIFO, is characterized by having elements inserted into one end and removed from the other end. • For example: a queue of people at a theater's box office. • The queue adaptor container is implemented in one of two ways: either as a deque or as a list.

 [17.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-17-638.jpg?cb=1421209622" \t "_blank" \o ) Note • As a general rule, always check which unique operations a certain container supports, and which common operations it disables. These should give you more than a hint regarding its optimal usage.

 [18.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-18-638.jpg?cb=1421209622" \t "_blank" \o ) Nays of containers

 [19.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-19-638.jpg?cb=1421209622" \t "_blank" \o ) Nays of Vector • When you find yourself, frequently inserting, removing elements in the middle of the vector. • If you think about inserting elements before the first element, then clearly vector is not the right choice because it doesn't even have a push\_front() operation.

 [20.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-20-638.jpg?cb=1421209622" \t "_blank" \o ) • Reallocation problem with vector. • Let's see how this can happen. Suppose you have a vector that has 256 elements: vector <int> vi(256); //vi contains 256 int's • At this point, you define an iterator that is pointing to the first element like this: for (vector<int>::iterator it= vi.begin();...) • Then, you call push\_back() to insert a new element: vi.push\_back(99); • From this moment, there's no guarantee that it is valid because the push\_back() call may have triggered reallocation

 [21.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-21-638.jpg?cb=1421209622" \t "_blank" \o ) Nays of List • When is list a bad choice? First and foremost, – When you need random access to elements. – If you need to sort the elements frequently. Although list does offer the sort()operation, sorting lists isn't as efficient as sorting vectors.

 [22.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-22-638.jpg?cb=1421209622" \t "_blank" \o ) Nays of Deque • When you find yourself, frequently inserting, removing elements in the middle of the deque. • Calling insert() in the middle of the deque invalidates all the iterators and references to its elements.

 [23.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-23-638.jpg?cb=1421209622" \t "_blank" \o "• Calling insert() at either end of the deque invalidates a...) • Calling insert() at either end of the deque invalidates all its iterators but has no effect on the validity of references to its elements. After the push\_front() call, the iterator it becomes invalid whereas the pointer p remains valid. • Similarly, calling erase() in the middle of the deque invalidates all the iterators and references to its elements. • Calling erase() at either end of the deque invalidates only the iterators and the references to the erased elements. deque<int> di; int p\* = &di[5]; deque<int>::iterator it=di.begin()+5; di.push\_front(8);

 [24.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-24-638.jpg?cb=1421209622" \t "_blank" \o ) Vector<bool> ?....... • The vector class template has a special template specialization for the bool type. • This specialization is provided to optimize for space allocation: In this template specialization, each element occupies only one bit (which is eight times less than the smallest type in C++: char).

 [25.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-25-638.jpg?cb=1421209622" \t "_blank" \o ) • The references to elements of a bool vector returned by the vector members are not references to bool objects, but a special member type which is a reference to a single bit, defined inside the vector<bool> class specialization as: class vector<bool>::reference { friend class vector; reference(); // no public constructor public: ~reference(); operator bool () const; // convert to bool reference& operator= ( const bool x ); // assign from bool reference& operator= ( const reference& x ); // assign from bit void flip(); // flip bit value. }

 [26.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-26-638.jpg?cb=1421209622" \t "_blank" \o ) Summary • If all you want is a "smart array" class that offers random access to elements, and perhaps the ability to append new elements to its end, then vector is the right choice. • When you need to insert or remove elements at either end of a container frequently, but you rarely use insertions and deletions in the middle, deque is a good choice. • If you don't need random access (say, "15 positions from the beginning" or "three positions before the current element") and you do need to insert new elements between existing elements frequently, then list is a good choice.

 [27.](http://image.slidesharecdn.com/effectivestl-130627021547-phpapp02/95/how-to-choose-best-containers-in-stl-c-27-638.jpg?cb=1421209622) • As associative containers, they are especially designed to be efficient accessing its elements by their key (unlike sequence containers, which are more efficient accessing elements by their relative or absolute position).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_