**Chaptor-01 Containers**

The STL has iterators, algorithms, and function objects. More powerful and flexible than arrays, they grow (and often shrink) dynamically, manage their own memory, keep track of how many objects they hold, bound the algorithmic complexity of the operations they support, and much, much more.

**Item 1. Choose your containers with care.**

C++ has variety of containers:

* The **standard STL sequence containers:** vector, string, deque, and list.
* The **standard STL associative containers**: set, multiset, map and multimap.
* The **nonstandard sequence containers** slist and rope.
* The **nonstandard associative containers** hash\_set, hash\_multiset, hash\_map,

and hash\_multimap.

* **vector<char> as a replacement for string.** There are certain conditions

under which such a replacement might make sense.

* **vector as a replacement for the standard associative containers. T**here are several times when vector can outperform the standard associative containers in both time and space.
* Several **standard non-STL containers,** including arrays, bitset, valarray, stack,

queue, and priority\_queue.

Case where arrays are preferable to STL containers. arrays can be used with

STL algorithms, because pointers can be used as array iterators. (Item 16)

bitset may be better than vector<bool> (Item 18)

vector, list, and deque offer the programmer different complexity trade-offs and should be used accordingly, vector is the type of sequence that should be used by default, list should be used when there are frequent insertions and deletions from the middle of the sequence, deque is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.

**Contiguous-memory containers vs node-based containers**

*Contiguous-memory containers* (also known as *array-based containers]* store them

elements in one or more (dynamically allocated) chunks of memory, each chunk

holding more than one container element. If a new element is inserted or an existing

element is erased, other elements in the same memory chunk must be shifted up or

down to make room for the new element or to fill the space formerly occupied by the

erased element. This kind of movement affects both performance and exception safety. The standard contiguous-memory containers are vector, string, and deque. The nonstandard rope is also a contiguous-memory container.

*Node-based containers* store only a single element per chunk of (dynamically

allocated) memory. Insertion or erasure of a container element affects only pointers to

nodes, not the contents of the nodes themselves, so element values need not be moved

when something is inserted or erased.

* For example: link list (list, slist), standard associative containers (set, multiset, map and multimap) and nonstandard hashed containers (hash\_set, hash\_multiset, hash\_map, and hash\_multimap).

**How to choose the container based of our requirements:**

* *If we want to insert a new element at an arbitrary position in the container:* If so, then need a sequence container, associative containers won't do.
* *If we care how elements are ordered in the container?* If not. a hashed container becomes a viable choice. Otherwise, you'll want to avoid hashed containers.
* *Must the container be part of standard C++?* If so, that eliminates hashed containers, slist, and rope.
* What *category of iterators we require?* If they must be random access iterators, then technically limited to vector, deque, and string, If bidirectional iterators are required, you must avoid slist.
* *Is it important to avoid movement of existing container elements when insertions or erasures take place?* If so, need to stay away from contiguous-memory containers.
* *Does the data in the container need to be layout-compatible with C*? If so,

we're limited to vectors.

* *Is lookup speed a critical consideration?* If so, look at hashed containers, sorted vectors, and the standard associative containers — probably in that order.
* *Do you mind if the underlying container uses reference counting? Use string.*
* Do *you need transactional semantics for insertions and erasures with reliably roll back insertions and erasures?* For a single element operation choose node-based container.If we need transactional semantics for multiple-element insertions need to choose list, because list is the only standard container that offers transactional semantics for multiple-element insertions.
* If we *need to minimize iterator, pointer, and reference invalidation?* If so,

you'll want to use node-based containers, because insertions and erasures on

such containers never invalidate iterators, pointers, or references (unless they

point to an element you are erasing). In general, insertions or erasures on

contiguous-memory containers may invalidate all iterators, pointers, and

references into the container.

* Would it be helpful to have a sequence container with random access iterators where pointers and references to the data are not invalidated if nothing is erased and insertions take place only at the ends of the container?

This is a very special case, but if it's your case, deque is perfect choice.

(Interestingly, deque's iterators may be invalidated when insertions are made

only at the ends of the container, deque is the only standard STL container

whose iterators may be invalidated without also invalidating its pointers and

references.)

**Item 2. Beware the illusion of container-independent code.**

The STL is based on generalization.

* Arrays are generalized into containers and parameterized on the types of objects they contain.
* Functions are generalized into algorithms and parameterized on the types of iterators they use.
* Pointers are generalized into iterators and parameterized on the type of objects they point to.

There are two types of containers, Sequence & associative container. Similar containers having similar functionality. For example:

Standard contiguous-memory containers offer random-access iterators, while standard node-based containers provide bidirectional iterators.

Sequence containers support push\_front and/or push\_back, while associative containers don't. Associative containers offer logarithmic-time lower\_bound, upper\_bound, and equal\_range member functions, but sequence containers don't.

Based on container type (sequence & associative) our code may vary time to time. For example:

Only sequence containers support push\_front or push\_back, and only associative containers support count and lower\_bound, etc. Even such basics as insert and erase have signatures and semantics that vary from category to category.

Let’s suppose If we want to write a code using sequence container (vector, deque, and list) Still there is variation to use them. There is no generalize concept for all sequence container.

We can’t use reserve() and capacity() for all sequence container because deque and list don't offer them. The presence of list also means need to give up operator[], and limit yourself to the capabilities of bidirectional iterators.

Also need to keep away from algorithms that demand random access iterators, including sort, stable\_sort, partial\_sort, and nth\_element.

All the above points concludes that we cant generalized the sequence container.

*"Generalized sequence container" where we can't call reserve, capacity, operator [], push\_front, pop\_front, splice, or any algorithm requiring random access iterators: a container where every call to insert and erase takes linear time and invalidates all iterators, pointers, and references.*

*Is that really the kind of container you want to use in your applications? I suspect not.*

The different containers are *different,* and they have strengths and weaknesses that vary in significant ways. They're not designed to be interchangeable.

But still wants to change the container type (custom container) use of typedefs

for container and iterator types. Hence, instead of writing this.

class Widget {...};

vector<Widget> vw;

Widget bestWidget;

… //give bestWidget a value

vector<Widget>::iterator i = // find a Widget with the

find(vw.begin(), vw.end(), bestWidget); // same value as bestWidget

write this:

class Widget {...);

typedef vector<Widget> WidgetContainer;

typedef WidgetContainer::iterator WCIterator;

WidgetContainer vw;

Widget bestWidget;

...

WCIterator i = find(vw.begin(), vw.end(), bestWidget);

Here we can simply to add a custom allocator to the container:

class Widget {... };

template<typename T> // see Item 10 for why this

SpecialAllocator{...} // needs to be a template

typedef vector<Widget, SpecialAllocator<Widget> *>* WidgetContainer;

typedef WidgetContainer::iterator WCIterator;

WidgetContainer vw; // still works

Widget bestWidget;

…

WCIterator i = find(vw.begin(), vw.end(), bestWidget); // still works

A typedef doesn't prevent a client from doing anything they couldn't already do. If we

want to limit client exposure to the container choices that we offered, we need classes.

If we replace one container type with another, hide the container in a class, and limit the amount of container-specific information visible through the class interface.

For example, if we need to create a customer list, don't use a list directly. Instead, create a CustomerList class, and hide a list in its private section:

class CustomerList {

private:

typedef list<Customer> CustomerContainer;

typedef CustomerContainer::iterator CCIterator;

CustomerContainer customers;

public:

... //limit the amount of list-specific

//information visible through

}; //this interface

Later point of time discovers that, we don't need to insert or erase customers from the middle of the list as often as we do anticipate, but we need to quickly identify the top 20% of your customers — a task tailor-made for the nth\_element algorithm. But nth\_element requires random access iterators. It won't work with a list. In that case, your customer "list" might be better implemented as a vector or a deque.

When we consider this kind of change, we still must check every CustomerList member function and every friend to see how they'll be affected in terms of performance and iterator/pointer/reference invalidation, etc.

But if we have done a good job of encapsulating CustomerList's implementation details, the impact on CustomerList clients should be small. *we can't write container-independent code, but they might be able to.*

**Item 3. Make copying cheap and correct for objects in containers.**

Containers hold objects, but not the ones you give them. Furthermore, when you get an

object from a container, the object you get is not the one that was in the container.

When you add an object to a container, what goes into the container is a *copy* of the object you specify. When you get an object from a container, what you set is a copy of what was contained. *Copy in, copy out. That's the STL way.*

An object is copied by using its copying member functions (*copy* constructor and *copy* assignment operator) For a user-defined class like Widget, these functions are traditionally declared like this:

class Widget {

public:

...

Widget(const Widget&); // copy constructor

Widget& operator=(const Widget&); // copy assignment operator

...

}

If we fill a container with objects where copying is expensive, then putting the objects into the container could be a performance bottleneck. Furthermore, if we have objects where "copying" has an unconventional meaning, putting such objects into a container will invariably lead to grief.

In the presence of inheritance, copying leads to slicing. That is, if we create a container of base class objects and try to insert derived class objects into it, the derivedness of the objects will be removed as the objects are copied (via the base class copy constructor) into the container:

vector<Widget> vw;

class SpecialWidget: // SpecialWidget inherits from

public Widget {...); // Widget above

SpecialWidget sw;

vw.push\_back(sw); // sw is copied *as a base class*

*II object* into vw. Its specialness

// is lost during the copying

The slicing problem suggests that inserting a derived class object into a container of base class objects is almost always an error.

An easy way to make copying efficient, correct, and immune to the slicing problem is to create containers of pointers instead of containers of objects. That is, instead of creating a container of Widget, create a container of Widget\*. Copying pointers is fast and nothing gets sliced when a pointer is copied.

Widget w[maxNumWidgets]; // create an array of maxNumWidgets

// Widgets, default-constructing each one

Using the STL instead of an array, we can use a vector that grows when it needs to:

vector<Widget> vw; // create a vector with zero Widget

// objects that will expand as needed

We can also create an empty vector that contains enough space for maxNumWidgets

Widgets, but where zero Widgets have been constructed:

vector<Widget> vw;

vw.reserve(maxNumWidgets);

Compared to arrays. STL containers are much more civilized. They create only as many objects as you ask for, they do it only when you direct them to, and they use a default constructor only when you say they should.

**Item 5. Prefer range member functions to their single-element**

**counterparts.**

class Widget {

string name;

int id;

public:

Widget(string \_nm,int \_id):name(\_nm),id(\_id){}

Widget(const Widget& widget):name(widget.name),id(widget.id){

cout << "Copy cons called\n";

}

Widget& operator=(const Widget& widget) {

if (this == &widget)

return \*this;

...

}

void display() { cout << "Name:: " << name << "ID:: " << id << endl; }

};

vector<Widget> wv;

Widget w1("Laptop", 10);

Widget w2("Watch", 20);

Widget w3("Tablet", 30);

Widget w4("Mouse", 40);

Widget w5("Pendrive", 50);

wv.reserve(10);

wv.push\_back(w1); //Not w1 goes inside the vector

wv.push\_back(w2); //,,

wv.push\_back(w3); //,,

wv.push\_back(w4); //,,

wv.push\_back(w5); //,,

Copy in-Copy out. But Why?

Suppose w1 goes directly inside the vector, what will happed it w1 goes out of scope, it will invalidate content inside the vector also. So, insertion make copy of object and that object will goes inside the vector.

Given two vectors, v1 and v2, what's the easiest way to make v1’s contents be the same as the second half of v2's?

v1.assign(v2.begin() + v2.size() /2, v2.end());

Range member functions are better than their single-element alternatives. A range member function is a member function that, like STL algorithms, uses two iterator parameters to specify a range of elements over which something should be done.

If we don’t like range function, then do something like this:

vector<Widget> v1, v2; // assume v1 and v2 are vectors of Widgets

v1.clear();

for ( vector<Widget>::const\_iterator ci = v2.begin() + v2.size() / 2; ci != v2.end(); ++ci)

v1.push\_back(\*ci);

But we should try to avoid the loop syntax. Its syntax complex then assign call and impose an efficiency penalty.

One way to avoid the loop to employ an algorithm instead that:

v1.clear();

copy(v2.begin() + v2.size() / 2, v2.end(), back\_inserter(v1 ));

But still more work than writing the call to assign. There is no loop is above code, one certainly exists inside copy. As a result, the efficiency penalty remains.

We can also replace copy function with insert range member functions. for example:

v1 .insert(v1 .end(), v2.begin() + v2.size() / 2, v2.end());

*Note: assign and insert are only equivalent if the vector is empty to begin with. Assign () will blow away anything that's already in the vector, then add the new elements. insert() doesn't touch any elements already in the vector.*

Finally, two reasons to prefer range member functions to their single-element counterparts:

* It's generally less work to write the code using the range member functions.
* Range member functions tend to lead to code that is clearer and more straightforward.

We are saying range method are more efficient then single element counterpart. Using standard sequence containers, application of single-element member functions makes more demands on memory allocators, copies objects more frequently, and/or performs redundant operations compared to range member functions that achieve the same end.

For example, suppose we like to copy an array of ints into the front of a vector (issues that arise when mixing STL containers and C APIs).

int data[numValues]; // assume numValues is

vector<int> v;

...

v.insert(v.begin(), data, data + numValues); // insert the ints in data into v at the front

Using iterative calls to insert in an explicit loop, it would probably look like this:

vector<int>::iterator insertLoc(v.begin());

for (int i = 0; i < numValues; ++i) {

insertLoc = v.insert(insertLoc, data[i]);

}

We need to carefully to save the return value of insert for the next loop iteration. If we didn't update insertLoc after each insertion, we'd have two problems:

* All loop iterations after the first would yield undefined behavior, because each insert call would invalidate insertLoc.
* Even if insertLoc remained valid, we'd always insert at the front of the vector (i.e., at v.begin()), and the result would be that the ints copied into v would end up in reverse order.

When we replace the loop with a call to copy, we get something like this:

copy(data. data + numValues, inserter(v, v.begin()));

Below are the range functions available for STL containers:

**Range construction.** All standard containers offer a constructor of this form:

*container::container(* Inputlterator begin, // beginning of range

Inputlterator end): //end of range

**Range insertion**. All standard sequence containers offer this form of insert:

void container::insert(iterator position, // where to insert the range

Inputlterator begin, // start of range to insert

InputIterator end); // end of range to insert

Associative containers use their comparison function to determine where elements go, so they offer a signature that omits the position parameter:

void container::insert(lnputIterator begin, Inputlterator end);

**Range erasure**. Every standard container offers a range form of erase, but the return types differ for sequence and associative containers. Sequence containers provide this,

iterator container::erase(iterator begin, iterator end);

while associative containers offer this

void container::erase(iterator begin, iterator end);

**Range assignment**. As I noted at the beginning of this Item, all standard sequence containers offer a range form of assign:

void container::assign(lnputIterator begin, Inputlterator end);

**Item 7. When using containers of newed pointers, remember to**

**delete the pointers before the container is destroyed.**

Containers in the STL are remarkably smart:

* They support iterators for both forward and reverse traversals
* During insertions and erasures, they take care of any necessary memory management; they report both how many objects they hold and the most they may contain (via size and max\_size, respectively)
* They automatically destroy each object they hold when they (the containers) are themselves destroyed.

But when the containers hold *pointers* to objects allocated with new, they're not right enough. Sure, a container of pointers will destroy each element it contains when it (the container) is destroyed, but the "destructor" for a pointer is a no-op! It certainly doesn't call delete.

void doSomething()

{

vector<Widget\*> vwp;

for (int i = 0; i < SOME\_MAGIC\_NUMBER; ++i)

vwp.push\_back(new Widget);

//Memory leak, if we don’t delete

for (vector<Widget\*>::iterator i = vwp.begin(); i != vwp.end(), ++i)

delete \*i;

}

There is two problem with above code:

* How we delete the widget if we use new style of *for loop* (for...each).
* Above code isn't exception safe. If an exception is thrown between the time vwp is filled with pointers and the time you get around to deleting them, you've leaked resources again.

Solution of first problem:

struct DeleteObject {

template<typename T>

void operator()(const T\* ptr) const

{

delete ptr;

}

}

void doSomething()

{

… // as before

for\_each(vwp.begin(), vwp.end(), DeleteObject());

}

Still the above code is not exception safe. This problem can be addressed in a variety of ways, but the simplest is probably to replace the container of pointers with a container of smart pointers, typically reference-counted pointers.

void doSomething()

{

Shared\_ptr<Widget> SPW; //shared pointer widget

vector<SPW> vwp;

for (int i = 0; i < SOME\_MAGIC\_NUMBER; ++i)

vwp.push\_back(new Widget); //create a SPW from widget\* then

//push to the vector.

….. // no Widgets are leaked here, not

// even if an exception is thrown

//in the code above

}

**Item 8. Never create containers of auto\_ptrs.**

Code attempt to compile below should not allowed:

Vector<auto\_ptr<Widget>> wv;

But why?

When we copy an auto\_ptr. ownership of the object pointed to by the auto\_ptr is transferred to the copying auto\_ptr and the copied auto\_ptr is set to NULL.

*to copy an auto\_ptr is to change its value:*

auto\_ptr<Widget> pw1 (new Widget); // pwl1points to a Widget

auto\_ptr<Widget> pw2(pw1); // pw2 points to pw1's Widget;

// pw1 is set to NULL. (Ownership

// of the Widget is transferred

// from pw1 to pw2.)

pw1 = pw2; // pw1 now points to the Widget

// again; pw2 is set to NULL

1

Consider the below piece of code, which is absolutely valid:

bool widgetAPCompare(const auto\_ptr<Widget>& lhs, const auto\_ptr<Widget>& rhs) {

return \*lhs < \*rhs; // for this example, assume that

} // operator< exists for Widgets

vector<auto\_ptr<Widget> > widgets; // create a vector and then fill it

// with auto\_ptrs to Widgets;

// remember that this should

// *not* compile!

Sort(widgets.begin(), widgets.end(), widgetAPCompare); // sort the vector

Problem is: sorting the vector may have changed its contents. One or more of the auto\_ptrs in

widgets may have been set to NULL during the sort.

**Item 9. Choose carefully among erasing options**

*Well documented in STL copy…*

**Item 12. Have realistic expectations about the thread safety of STL**

**containers.**

STL Container, we can hope for below point but not decently can expect. Some implementations offer these guarantees, but some do not.

* Multiple readers are safe. Multiple threads may simultaneously read the contents of a single container, and this will work correctly. Naturally, there must not be any writers acting on the container during the reads.
* Multiple writers to different containers are safe. Multiple threads may simultaneously write to different containers.

Writing multithreaded code is hard, and many programmers wish that STL implementations were completely thread safe out of the box. Programmers can achieve the complete thread safety on STL container, but this is not easy task. Consider the following ways a library might try to implement such comprehensive container thread safety:

* Lock a container for the duration of each call to its member functions.
* Lock a container for the lifetime of each iterator it returns (via, e.g.., calls to begin or end).
* Lock a container for the duration of each algorithm invoked on that container.

Now consider the following code. It searches a vector<int> for the first occurrence of

the value 5, and, if it finds one, changes that value to 0.

vector<int> v;

vector<int>::iterator first5(find(v.begin(), v.end(), 5)); // Line 1

if (first5 !=v.end()){ // Line 2

\*first5 = 0; // Line 3

}

In a multithreaded environment, it's possible that a different thread will modify the data in v immediately after completion of Line 1 (by performing an insertion that caused the vector to

reallocate its underlying memory. That would invalidate all the vector's iterators).

Similarly, the assignment to \*first5 on Line 3 is also unsafe, because another thread might execute between Lines 2 and 3 (other part of code) in such a way as to invalidate firsts, by erasing the element it points.

For the code above to be thread safe, v must remain locked from Line 1 through Line 3.

vector<int> v;

…

getMutexFor(v);

vector<int>::iterator first5(find(v.begin(), v.end(), 5));

if (first5 != v.end()) { // this is now safe

\*first5 = 0; // so is this

}

releaseMutexFor(v);

A more object-oriented solution is to create a Lock class that acquires a mutex in its constructor and releases it in its destructor.

template<typename Container> // skeletal template for classes

class Lock { // that acquire and release mutexes

public: // for containers;

…

Lock(const Containers container): c(container)

{

getMutexFor(c); // acquire mutex in the constructor

}

~Lock()

{

releaseMutexFor(c); // release it in the destructor

}

private:

const Container& c;

};

The idea of using a class (like Lock) to manage the lifetime of resources (such as

mutexes) is generally known as *resource acquisition is initialization*. Now our code looks like:

vector<int> v;

…

{ // create new block;

Lock<vector<int> > lock(v); // acquire mutex

vector<int>::iterator first5(find(v.begin(), v.end(), 5));

if (first5 != v.end()) {

\*first5 = 0;

}

} // close block, automatically

// releasing the mutex