

More on STL ☹️

Erasing from Vectors

- **V.erase(iterator Iter)**
 - Erases the element at the position denoted by Iter, shifting the elements after it back to fill the space left by the deleted elements. ***The size of the vector will decrease by 1.*** Returns the next valid iterator. If you erase the last element, it will point to .end().
- **V.erase(iterator First, iterator Last)**
 - Erases elements starting at First and stopping at the element just before Last, shifting elements after the range back to fill the space. ***The size of the vector will decrease by the distance between First and Last.***

```
vector<int> V;  
for (int i = 1; i <= 6; i++) {  
    V.push_back(i);  
} // Line 1  
  
V.erase(V.begin() + 2); // Line 2  
  
V.erase(V.begin(), V.begin() + 2); // Line 3  
  
V.pop_back(); // Line 4  
  
V.erase(V.begin(), V.end()); // Line 5
```

After Line 1:

1	2	3	4	5	6
---	---	---	---	---	---

After Line 2:

1	2	4	5	6
---	---	---	---	---

After Line 3:

4	5	6
---	---	---

After Line 4:

4	5
---	---

After Line 5:

empty

Conditionally Erasing elements in a loop

```
vector<int> res;

// add elements here.

vector<int>::iterator it = res.begin();

for ( ; it != res.end(); ) {
    if (condition) {
        it = res.erase(it);
    } else {
        ++it;
    }
}
```

Do NOT do the following:

```
for( ; it != res.end(); it++)
{
    it = res.erase(it);
}
```

Inserting elements to a List

```
// declaring list  
list<int> list1;
```

```
// using assign() to insert multiple number  
list1.assign(3,2);
```

creates 3
occurrences of "2"
from list1.begin().
Overwrites all
existing data.

```
// initializing list iterator to beginning  
list<int>::iterator it = list1.begin();
```

```
// iterator to point to 3rd position  
advance(it, 2);
```

```
// using insert to insert 1 element at the 3rd position  
// inserts 5 at 3rd position. List now becomes 2 2 5 2.  
list1.insert(it,5);
```

Return value is an
iterator that
points to 5. (the
newly inserted
element)

More list stuff

```
std::list<int> mylist;
std::list<int>::iterator it;

// set some initial values:
for (int i=1; i<=5; ++i) mylist.push_back(i); // 1 2 3 4 5

it = mylist.begin();
++it;          // it points now to number 2

mylist.insert (it,10);      // 1 10 2 3 4 5

// "it" still points to number 2
mylist.insert (it,2,20);    // 1 10 20 20 2 3 4 5

--it;           // it points now to the second 20

std::vector<int> myvector (2,30); // vector of two 30
mylist.insert (it, myvector.begin(),myvector.end());
// 1 10 20 30 30 20 2 3 4 5
```

Inserting into Maps

```
std::map<char, int> mymap;
```

```
mymap.insert ( pair<char,int>('a',100) );  
mymap.insert ( pair<char,int>('z',200) );
```

First Map insert
version. 1
parameter

```
pair<map<char,int>::iterator, bool> ret;  
ret = mymap.insert (pair<char,int>('z',500) );  
if (ret.second==false) {  
    std::cout << "element 'z' already existed";  
    std::cout << " with a value of " << ret.first->second << "\n";  
}
```

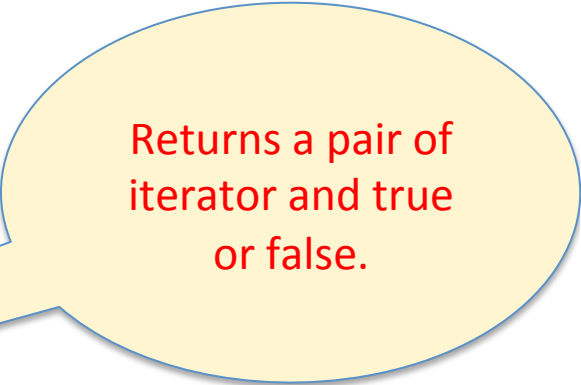
Return value is a
pair of iterator,bool

```
map<char,int>::iterator it = mymap.begin();  
mymap.insert (it, std::pair<char,int>('b',300)); // max efficiency inserting  
mymap.insert (it, std::pair<char,int>('c',400)); // no max efficiency inserting
```

Second insert
version with hint
position

```
// third insert function version (range insertion):  
std::map<char,int> anothermap;  
anothermap.insert(mymap.begin(),mymap.find('c'));
```

Sets



Returns a pair of
iterator and true
or false.

```
set<int> myset;
set<int>::iterator it;
pair<set<int>::iterator,bool> ret;

// set some initial values:
for (int i=1; i<=5; ++i) myset.insert(i*10);
    // set: 10 20 30 40 50

ret = myset.insert(20);           // no new element inserted

if (ret.second==false)
    it=ret.first;  // "it" now points to element 20

myset.insert (it,25);             // max efficiency inserting
myset.insert (it,24);             // max efficiency inserting
myset.insert (it,26);             // no max efficiency inserting

int myints[]={5,10,15};          // 10 already in set, not inserted
myset.insert (myints,myints+3);  // 5 10 15 20 24 25 26 30 40 50
```


Prefer List when

- you need **constant-time insertions/deletions** from the list (such as in real-time computing where time predictability is absolutely critical)
- you don't know how many items will be in the list. With arrays, you may need to re-declare and copy memory if the array grows too big
- you don't need random access to any elements
- you want to be able **to insert items in the middle** of the container.

Prefer Vector when

- you **need indexed/random access** to elements
- you know the number of elements in the array ahead of time so that you can allocate the correct amount of memory for the vector
- you need **speed when iterating through all the elements in sequence**. You can use pointer math on the array to access each element, whereas you need to lookup the node based on the pointer for each element in linked list, which may result in page faults which may result in performance hits.
- **memory is a concern**. Filled arrays take up less memory than linked lists. Each element in the array is just the data. Each linked list node requires the data as well as one (or more) pointers to the other elements in the linked list.

Some Common Mistakes with STL

- Template of templates

```
stack<vector<int>>> GoneWrong; // need space: "> >"
```

- Same algorithm, different container

```
sort(Marmalade.begin(), Jelly.end()) // crash!
```

- Right iterator, wrong container

```
list<int>::iterator i = listone.begin();  
listtwo.erase(i); // i doesn't point into listtwo
```

Common Mistakes (cont.)

- Invalid iterator

```
vector<int>::iterator i = GrandDayOut.begin();  
GrandDayOut.push_back(1); // potential realloc  
foobar(*i); // uh-oh! i might be invalid
```

- Adding elements with subscript op

```
vector<char> v1;  
v1[0] = 'W'; // crash!  
v1.push_back('W'); // the right way
```

Common Mistakes (cont.)

- Sorting problems - e.g. lookups fail, a set gets duplicates
- Usually a problem with $op <$ or $op ==$
- Rules you should follow while implementing
 - if $x < y$ is true $\Rightarrow y > x$ is true
 - if $x < y$ & $y < z$ are true, $x < z$ is always true
 - if $x == y$, then $x < y$ is always false
 - if $!(x < y)$ and $!(y < x)$, $x == y$

Hiding the Angle Brackets

- Not pretty

```
// This is hard to read
```

```
map<string, int> MyMap;
```

```
MyMap.insert(pair<string, int>("abc", 31));
```

```
map<string, int>::iterator i = MyMap.find("abc");
```

```
pair<string, int> pr = *i;
```

```
pr.first; // "abc"
```

```
pr.second; // int 31
```

Hiding the Angle Brackets (cont.)

- Typedefs are your friend

```
// Tuck these away in a header file
typedef map<string, int> MyMap;
typedef pair<string, int> MyMapPair;
typedef MyMap::iterator MyMapItr;
```

```
// Same code, but no more angle brackets
MyMap map1;
map1.insert(MyMapPair("abc", 31));
MyMapItr i = map1.find("abc");
MyMapPair pr = *i;
```

Storing Pointers in Containers

- Container will *not* delete the objects when the container goes away – only the pointers.
- Will sort on pointer, not what it contains, ***unless you tell it otherwise***

```
deque<int*> myqueue;           // sorted by pointer
Sort(myqueue.begin(), myqueue.end());
```

```
bool intpLess(const int* a, const int* j)
    { return (*a < *b); } // sorted by int value not pointer
sort(Walkies.begin(), Walkies.end(), intpLess);
```

Vector Tips

- Use `reserve()` or a parameterized constructor to set aside space when vector size is known in advance
- Can I safely take the address of a vector element, e.g. `&v[21]`?
 - According to Standard, no. According to practice, yes. Standard expected to adjust.
- Trimming unused space

```
v.swap(vector<T>(v)); // vector, swap thyself!
```


Checking for Empty Container

- Write `if (c.empty())` instead of `if (c.size() == 0)`
 - `empty()` is guaranteed constant time
 - `size()` is not guaranteed constant time (it usually is, but not always, e.g. lists)

Copying Containers

- The wrong way; `copy()` cant add elems

```
copy(v.begin(), v.end(), nv.begin()); // uh-oh
```

- Better and best

```
nv.resize(v.size()); // size of nv matches v  
copy(v.begin(), v.end(), nv.begin());
```

```
copy(v.begin(), v.end(), back_inserter(nv));
```

Algorithms that Remove Elems

- Algorithms by themselves can't insert or delete elements, so they move them!
- `unique()` moves unique elems to the front and returns an iter to the new “end” of the container
- `remove()` similarly moves the “unremoved” elems to the front and returns an iter to the new “end”

Removing Elems (cont.)

- To actually get rid of the extra elems, you must call `erase()`

```
// Removes duplicates and shrinks the container
```

```
v.erase(unique(v.begin(), v.end()), v.end());
```

```
// Removes the given elements and shrinks v
```

```
v.erase(remove(v.begin(), v.end(), 1), v.end());
```

Iterator Troubleshooting

- Iter dereferenceable? (end() is not)
- Container invalidated the iter?
- Pair of iters a valid range? First really before last?
- Iter pointing to the right container?
- Pair of iters pointing to the same container?

Vectors vs C-style arrays

- Prefer vector or deque over arrays
- Don't have to know size in advance
- Don't have to keep track of size separately
- Little loss of efficiency
- Vector works well with legacy code that expect arrays

The “Key” Requirement

- Once a key has been inserted into a container, the key should not change
- Can't be enforced internally by the STL
- Example: key is a filename; comparison is based on file contents

Dynamic Memory Review

Memory Allocation Errors

- Memory Allocation is Error Prone
 1. Memory Leaks
 2. Premature Free
 3. Double Free
 4. Wild Frees
 5. Memory Smashing

1- Memory Leaks

- Memory leaks are objects in memory that are no longer in use by the program but that ***are not freed***.
- This causes the application to use excessive amount of heap until it runs out of physical memory and the application starts to swap slowing down the system.
- If the problem continues, the system may run out of swap space.
- Often server programs (24/7) need to be “rebounded” (shutdown and restarted) because they become so slow due to memory leaks.

Memory Leaks

- Memory leaks is a problem for long lived applications (24/7).
- Short lived applications may suffer memory leaks but that is not a problem since memory is freed when the program goes away.
- Memory leaks is a “slow but persistent disease”. There are other more serious problems in memory allocation like premature frees.

Memory Leaks

```
int * ptr;  
ptr = new int;  
*ptr = 8;  
... Use ptr ...  
ptr = new int;  
// Old block pointed by ptr  
// was not deallocated.
```

2- Premature Frees

- A premature free is caused when ***an object that is still in use by the program is freed.***
- The freed object is added to the free list modifying the next/previous pointer.
- If the object is modified, the next and previous pointers may be overwritten, causing further calls to new/delete to crash.
- Premature frees are difficult to debug because the crash may happen far away from the source of the error.

Premature Frees

```
int * p = new int;
p = 8;
delete p; // delete adds object to free list
          // updating header info

...
*p = 9; // next ptr will be modified.
... Do something else ...
int *q = new int;
    // this call or other future new/delete
    // calls will crash because the free
    // list is corrupted.
    // It is a good practice to set p = NULL
    // after delete so you get a SEGV if
    // the object is used after delete.
```

Mitigating the problem

```
int * p = new int;  
p = 8;  
delete p; // delete adds object to free list  
p = NULL; // Set p to NULL so it cannot be used  
*p = 9; // You will get a SEGV. Your program  
        will crash and you will know that you have  
        already freed p.
```

3 - Double Free

- Double free is caused by freeing memory that is already free.
- This can cause the object to be added to the free list twice corrupting the free list.
- After a double free, future calls to new/delete may crash.

Double Free

```
int * p = new int;  
delete p; // delete adds object to free list  
.. Do something else ....  
delete p; // deleting the object again  
           // overwrites the next/prev ptr  
           // corrupting the free list  
           // future calls to free/malloc  
           // will crash  
  
// Also to prevent this problem you may set  
p to NULL after free.
```

4- Wild Frees

- Wild frees happen when a program attempts to free a pointer in memory that was not returned by new.
- Since the memory was not returned by new, it does not have a header.
- When attempting to free this non-heap object, the free may crash.
- Also if it succeeds, the free list will be corrupted so future new/delete calls may crash.

Wild Frees

- Also memory allocated with *malloc()* should only be deallocated with *free()* and memory allocated with *new()* should only be deallocated with *delete()*.
- Wild frees are also called “free of non-heap objects”.

Wild Frees

```
int q;  
int * p = &q;  
  
free(p) ;  
    // p points to an object without  
    // header. Free will crash or  
    // it will corrupt the free list.
```

Wild Frees

```
char * p = new char[100];  
p=p+10;  
  
free(p) ;  
// p points to an object without  
// header. Free will crash or  
// it will corrupt the free list.
```

5- Memory Smashing

- Happens when less memory is allocated than the memory that will be used.
- This causes overwriting the header of the object that immediately follows, corrupting the free list.
- Subsequent calls to new/delete may crash
- Sometimes the smashing happens in the unused portion of the object causing no damage.

Memory Smashing

```
char * s = new char[8];  
strcpy(s, "hello world");  
  
// We are allocating less memory for  
// the string than the memory being  
// used. Strcpy will overwrite the  
// header and maybe next/prev of the  
// object that comes after s causing  
// future calls to new/delete to crash.  
// Special care should be taken to also  
// allocate space for the null character  
// at the end of strings.
```

Other Memory Related Errors

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables

Dereferencing Bad Pointers

- The classic `scanf` bug (only relevant to C)

```
scanf("%d", val);
```

- OR

```
int *p;  
  
*p = 10;
```

Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
    int *y = new int[N];
    int i, j;

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            y[i] += A[i][j]*x[j];
    return y;
}
```

Overwriting Memory

- Off-by-one error

```
int **p;  
  
p = new int*[N];  
  
for (i=0; i<=N; i++) {  
    p[i] = new int[M];  
}
```

Overwriting Memory

- Not checking the max string size

```
char s[8];  
int i;  
  
gets(s); /* reads "123456789" from stdin */
```

- Basis for classic buffer overflow attacks
 - 1988 Internet worm
 - Modern attacks on Web servers
 - AOL/Microsoft IM war

Overwriting Memory

- Referencing a pointer instead of the object it points to

```
int* size;  
*size++;
```

- Remember – both * and ++ have the same precedence which implies?????

Overwriting Memory

- Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {  
    while (*p && *p != val)  
        p += sizeof(int);  
  
    return p;  
}
```

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {  
    int val;  
    return &val;  
}
```