Overview
Standard C++ Library

Concepts
&
Examples

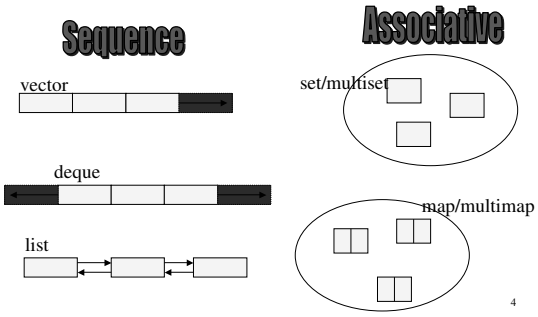
Question: Why should I write
***Stack*, *List*, *Set*, *Sort*, *Search*, *etc*?**

Answer: don't!!!
use the C++ Standard Library.
Heart of the library: STL

The standard C++ library has:

- ***Containers***
- ***Iterators*** to "run through" the containers
- ***Algorithms*** that work on containers: **find, sort, etc.**
- container ***adapters***: stack, queue, priority_queue
- iterator adapters
- strings

STL Containers



STL Containers

adapters:

stack

queue

priority_queue

5

containers include

- *sequential*: position depends on time and place of insertion but not on the value of item -- vector, deque, list
- *associative*: position depends on its value -- set/multiset, map/multipmap
- *adapters*: stack, queue, priority_queue
- *string*

6

containers

- grow as needed -- good to manage data, as used in typical programs:

  programs start with an empty container, read or compute data, store it, access it, store some more…
- built in arrays don't grow
- all 3 sequential containers have **push_back**

7

Some functions for sequential containers:

- push_back(), pop_back()
- push_front(), pop_front()
- size()
- begin() -- points to first item in vector
- end() -- points one past end of vector
- iterator: pointer to an item in the vector

8

C++ strings are containers: better than C strings

- Memory managed for you:
  C++ ==> ***string s("dog")***
  C ==> ***s = new char[4]; strcpy(s, "dog")***
- easier to use
  C++ ==> ***s <= t***
  C ==> ***strcmp(s, t) <= 0***
- less error prone than C strings

9

Iterators

- step through a container, independent of internal structure
- small but common interface for any container
- every container has its own iterator
- interface for iterators almost the same as pointers: *, ++, --

10

Operations

- operator *
- operator++ and operator--
- operators == and !=
- operator =
- begin()
- end() (past the end iterator)
- rbegin()
- rend()

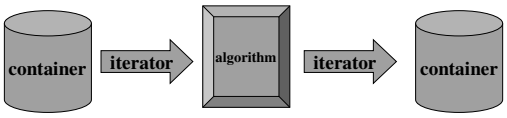
11

const and non const iterators:

- ***const_iterator*** - items in the container may not be modified and must only be used with constant member functions
- ***iterator*** - permits modification of the items in the container
- ***reverse_iterator*** – backward traversal
- ***const_reverse_iterator***

12

Algorithms



Algorithms process the elements of a container
Algorithms use iterators

Separation of data and operations:
an algorithm can be written once and work with arbitrary containers because iterators have a common interface.

13

STL and OO

- OO says: encapsulate data and the operations on that data
- The STL contradicts OO philosophy
- reason: in principle, you can combine every kind of container with every kind of algorithm
- result: small, flexible framework that is type independent

14

Ranges

- all algorithms process a range
- the range might be the whole container, but not necessarily
- pass beginning and end of range as two separate arguments
- programmer must ensure that the range is valid
- passing an invalid range is undefined

15

open-ended range

- every algorithm processes a half-open range
- a range includes the beginning position but excludes the end position
- [begin, end)
- advantages:
  - simple end criterion for loops
  - avoids special handling for empty ranges

16

Some algorithms & ranges

```
list<int> mylist, secondlist;
vector<int> vec;
for (int I = 0; I < 100; ++I) mylist.push_back(I);

list<int>::const_iterator ptr =
        find(mylist.begin(), mylist.end(), 99);

cout << *max_element(mylist.begin(), mylist.end());

if ( equal(mylist.begin(), mylist.end(), secondlist.begin()) ) {
  ...
}
copy( mylist.begin, mylist.end(),      // source
        vec.begin() );                 // destination
```


Efficiency of vector search

- Let's search a vector using 3 difference approaches
- for each, take the average of 100 searches
- execute on Gateway x86 laptop, 1GHz processor, 160 mg RAM
- Running RedHat linux 7.1
- gcc 2.96


```
const int MAX = 1000000;
const int LOOP_COUNT = 100;

int main() {
   vector<int> vec;
   for (int i = 0; i < MAX; ++i) {
     vec.push_back( 17 );
   }
   vec.push_back( 99 );

   double sum = 0;
   for (int i = 0; i < LOOP_COUNT; ++i) {
     time_t start = clock();
        unsigned int number = my_locate(vec);
     time_t end = clock();
     sum += (static_cast<double>(end)-start)/1000;
   }
   cout << "Avg time: " << sum/LOOP_COUNT << endl;
}
```

Efficiency?


```
int my_locate(const vector<int> & vec) {
   for (int i = 0; i < vec.size(); ++i) {
      if (vec[i] == 99 )  return vec[i];
   }
   return 0;
}
```

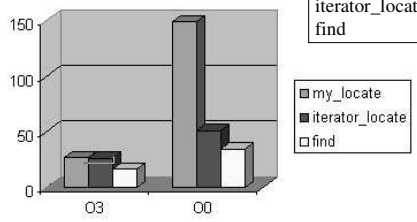
Which is faster?

```
int iterator_locate(const vector<int> & vec) {
   vector<int>::const_iterator ptr = vec.begin();
   while ( ptr != vec.end() ) {
      if (*ptr == 99 ) { return *ptr; }
      ++ptr;
   }
   return 0;
}
```

```
vector<int>::const_iterator ptr =
     find(vec.begin(), vec.end(), 99);
```

Timing results

	O3	O0
my_locate	27.1	148.5
iterator_locate	26.4	51.5
find	17	34.6



21

More efficiency: value vs const/ref

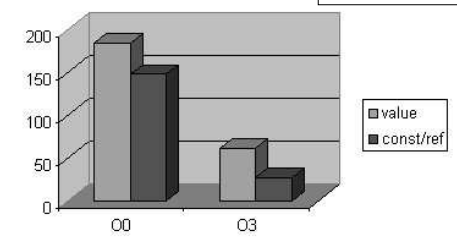
```
int my_locate(const vector<int> & vec) {
   for (int i = 0; i < vec.size(); ++i) {
      if (vec[i] == 99 ) return vec[i];
   }
   return 0;
}

int your_locate(vector<int> vec) {
   for (int i = 0; i < vec.size(); ++i) {
      if (vec[i] == 99 ) return vec[i];
   }
   return 0;
}
```

22

Timing results

	O0	O3
value	183.9	61.5
const/ref	148	27



23

Conclusions about efficiency

- Hard to beat STL built in algorithms, like *find*, *copy*, iterator adapters
- Passing large structures by value is costly, difference would be more dramatic if vector contained large structure, or strings
- Level O3 optimization on gcc makes a big difference!

24

Iterator adapters

- Iterators are pure abstractions
- anything that behaves like an iterator, is an iterator
- can write classes that have interface of iterators but do something different
- STL provides several predefined iterators:
  - insert iterators
  - stream iterators
  - reverse iterators

25

Insert iterator

- Inserters allow algorithms to operate in insert mode, rather than overwrite mode
- solve a problem: allow algorithms to write to a destination that doesn't have enough room – grow the destination
- three kinds that insert – at front, at end, or at a given location

26

```
int main() {
  list<int>          mylist;
  vector<int>        vec;
  deque<int>         deck;
  set<int>           myset;

  for (int I = 1; I <= 10; ++I) {
    mylist.push_back(I);
  }
  copy(mylist.begin(), bylist.end(),      // source
         back_inserter(vec) );            // destination
  copy(mylist.begin(), mylist.end(),      // source
       front_inserter(deck);              // destination
  copy(mylist.begin(), mylist.end(),      // source
       inserter(myset, myset.begin() )) // destination
}
```

what does vec, deck & myset look like?

27

How predefined inserters work

- back_inserter is like push_back() – only makes sense if container defines it
- front_inserter is like push_front() -- note that this reverses the order of inserted items – only makes sense if container defines push_front
- inserter calls insert() member function with the new value and the position (for associative containers the position is a hint to start search)

28

Stream iterators

- read from and write to a stream
- a stream is an object that represents I/O channels
- lets input from keyboard behave as a collection, from which you can read
- can redirect output to a file or screen

29

What does this code do?

```
int main() {
  vector<string> vec;
  copy ( istream_iterator<string>(cin),  // start of source
         istream_iterator<string>(),     // end of source
         back_inserter(vec) );           // destination

  sort( vec.begin(), vec.end() );

  unique_copy(vec.begin(), vec.end(),   // source
       ostream_iterator<string>(cout, "\n") );  // destination
}
```

30

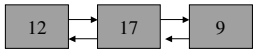
inserting & erasing vector/list elements

- insert(pos, elem) – inserts a copy of elem at position pos, returns position of new element
- insert(pos, n, elem) – inserts n copies of elem at pos (returns nothing)
- insert(pos, beg, end) – inserts, starting at pos, copy of elements [beg, end) – (returns nothing)
- erase(pos) -- removes the element at pos, returning position of next element
- erase(beg, end) – removes elements from [beg, end) & returns position of next element

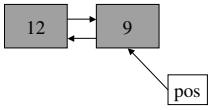
31

Erasing from a list

list<int>::iterator pos = mylist.begin();
++pos;



pos = mylist.erase(pos);



32

This code has a memory leak

```
vector<Student *> vec;
vec.push_back( new Student("Anakin") );

vec.erase( vec.begin() );
```

33

Function objects

sorting and searching
when it ain't easy,
like with pointers

34

pointers

- containers can be used to store user-defined objects as well as primitive types
- sort, find and find_if will work if **<** and **==** are defined
- with pointers, they still may not work, even with **<** and **==**
- enter function objects

35

Function object

- a function object (or functor) is an object that has operator() defined
- for example:

```
class FunctionObject {
public:
   void operator() () { … }
};
FunctionObject fo;
```

36

Advantages of function object

- can have state
- has its own **type** – can be passed to a template
- Usually faster than a function pointer
- to see how they work, let's write our own *find_if*!


```
#include <vector>
#include <algorithm>
class Less50 {
public:
  Less50()  { cout << "default" << endl; }
  bool operator() (int x) const { return x < 50; }
};

vector<int>::const_iterator find_if(const vector<int> & v, const Less50 obj) {
  vector<int>::const_iterator ptr = v.begin();
  while (ptr != v.end() ) { if (obj(*ptr)) return ptr;  ++ptr; }
  return ptr;
}

void fill(vector<int> & n) { }
int main() {
  vector<int> items;   fill(items);
  vector<int>::const_iterator ptr = find_if(vec, Less50() );
  if ( ptr != vec.end() ) cout << "Yes" << endl; else cout << "No" << endl;
}
```

Less50 is a function object

write our own find_if

calls default constructor to create an instance of Less50


```
#include <vector>
#include <algorithm>

class Less50 {
public:
  bool operator() (int x) const { return x < 50; }
};

void find_less50(const vector<int> & v) {
  vector<int>::const_iterator ptr = find_if(v.begin(), v.end(), Less50() );
  if (ptr != v.end() ) cout << the first value < 50 is: " << *ptr << endl;
}

void fill(vector<int> & n) { }

int main() {
  vector<int> items;   fill(items);
  find_less50(items);
}
```

Less50 is a function object

this find_if is in STL

Function objects can have state

- can store local values
- can provide auxiliary functions
- assume the following function:

```
template <class T>
inline void PRINT_ELEMENTS(const T &coll,
         const char * optcstr="") {
  typename T::const_iterator pos;
  std::cout << optcstr;
  for (pos = coll.begin(); pos != coll.end(); ++pos) {
     std::cout << *pos << ' ';
  }
  std::cout << std::endl;
}
```

```
class AddValue {
public:
   AddValue(int v) : theValue(v) {}
   void operator() ( int &elem) const {
      elem += theValue;
   }
private:
   int theValue;
};

int main() {
   list<int> coll;
   for (int i = 0; i < 10; ++i) coll.push_back(i);
   PRINT_ELEMENTS(coll, "Initial values: ");
   for_each( coll.begin(), coll.end(), AddValue(10) );
   PRINT_ELEMENTS(coll, "Values after adding 10: ");
   for_each(coll.begin(), coll.end(),
            AddValue(*coll.begin()));
   PRINT_ELEMENTS(coll, "Values plus first value: ");
}
```

using state

41

```
#include <vector>
#include <algorithm>
class Student {
public:
  string getName() const { return name; }
private:
  string name;
};
class StudentLess {
public:
  bool operator()(const Student * left, const Student * right) const {
    return left->getName() < right->getName();
};
int main() {
  vector<Student*> items;
  sort(items.begin(), items.end(), Less() );
}
```

42

```
#include <vector>
#include <algorithm>
class Student {
public:
  string getName() const { return name; }
private:
  string name;
};
class StudentEqual {
public:
  StudentEqual(const string & n) : name(n) { }
  bool operator()(const Student * left, const Student * right) const {
    return name ==  right->getName();
  }
private:
  string name;
};
```

43

Using the saved state

```
int main() {
  vector<Student *> vec;
   // fill the vector;
   StudentEqual ob("Mary");
   vector<Student *>const_iterator ptr =
                              find_if(vec.begin(), vec.end(), ob);
    if (ptr != vec.end() ) {
      cout << *(*ptr) << endl;
    }
}
```

44

Standard library exercises

- Design an experiment to show *binary_search* faster than *find*
- write a program that inserts some items into a vector and prints the list sorted by name (use a fn obj)
- extend your items program to find a name read from the terminal
- extend to read the items from a file

45

Namespaces

Used by Standard C++ Library

46

Name clashes

- current software trends: build libraries, modules, components
- when combined, names can clash
- Namespaces solve this problem
- example:

```
namespace NS {
   class Student {};
   void print();
   int x;
}
```

47

3 ways to access namespace items

- with the scope qualifier: *NS::x;*
- qualifier with using: using *NS::x;*
- the using directive:
  using namespace NS;

48

argument dependent name lookup

- Functions don't have to be qualified if one or more arguments are from the namespace

```
NS::Student stu;
print(stu);
cout << stu << end;
```

49

C++ Standard library -- std

- All identifiers are in namespace std
- Some compilers enforce use of std namespace
- gcc's C++ compiler does not enforce std until 3.0.4

```
std::cout << std::hex << 3.4 << std::endl;
```

50

Value semantics

containers contain the values of the objects, not the objects themselves

51

value vs reference semantics

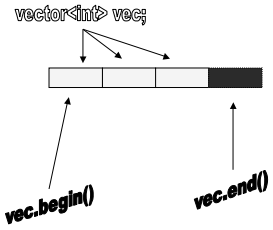
- all containers create internal copies of item & return copies
- container items are equal but not identical to objects you put in
- if you modify a container item, you modify a copy not the original
- to get reference semantics, use pointers

52

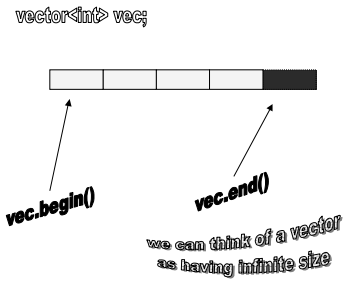
An ordered collection: vector

a random-access container
(not necessarily sorted)

A vector is a block of data with sub-blocks of size *type* for vector<*type*>:



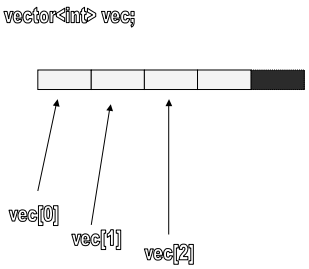
push_back() adds a sub-block to the end of the block:



Vector efficiency:

- good performance for random access
- good performance when append/delete at end
- push_back() & pop_back() on average: O(1)
- push_front() and pop_front() are not supported
- insert(begin()) and delete(begin()) are O(n)
- if inserting or deleting from front or middle, switch to different container

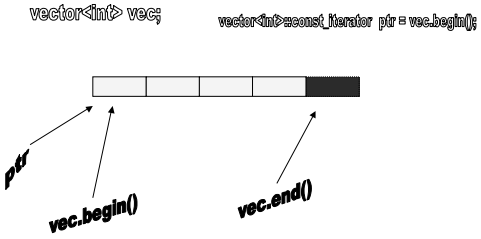
random access of items: use array notation:



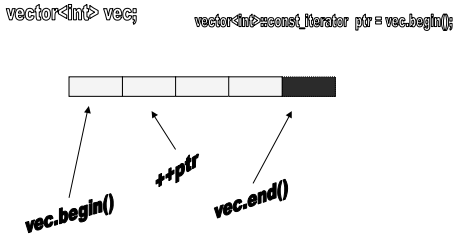
array notation with vectors:

- familiar
- only way to access non-sequentially
- but, library algorithms use iterators
- iterators can be written to work on any container – can then switch if necessary

An iterator is a pointer to a subblock:



We can move the iterator:



size/capacity

- size() – returns actual number in container
- capacity() – returns the current "amount of room"
- max_size() – max number of elements a container might contain; implementation defined
- empty()

61

Reserve

Can avoid capacity doubling if I know how much room I will need:

```
std::vector<int> vec;
vec.reserve(100);  // reserve memory for 100 ints
```

after the vector contains 100 items, the capacity doubles.

cannot use reserve to shrink the vector.
to shrink:
use erase, assignment

62

setting capacity with constructor

The next statement creates space for 100 items, using the default constructor to initialize the memory:

std::vector<T> vec(100);

It's similar to an array:
T items[100];

63

What's the output?

```
vector<int> vec1(100);
vector<int> vec2;
vec2.reserve(100);

vec1.push_back(7);
vec2.push_back(7);

cout << vec1.capacity() << endl;
cout << vec2.capacity() << endl;
```

64

deque

another random access, sequential container

65

comparing deque and vector

- like a vector, but open at both ends
- fast access at begin and end of deque
- usually implement as two blocks that grow
- like a vector, supports indexing, but it's usually more expensive than a vector
- Not used often

66

list

a sequential container

67

operations

- it's an ordinary doubly-linked list
- insert/erase in O(1)
- supports push_front, push_back, pop_front, pop_back
- only way to access the nth element is to iterate from beginning
- does not support [] for same reason that vector does not support push_front()

68

iterator invalidation

- in general, iterators that refer to container elements are valid as long as the elements stay in the same place!
- operations can cause elements to move
- if insertion causes a vector to grow, then a previously defined iterator is invalid
- in contrast, a list iterator never becomes invalid as long as elements exist

69

list: inserting & removing

c.insert(pos, elem);	inserts at iterator position *pos*, a copy of *elem,* and returns the position of the new element
c.push_back(elem);	appends a copy of *elem* at end
c.pop_back();	removes last element (void fn)
c.erase(pos);	removes the element at iterator position *pos* and returns the position of the next element
c.clear();	removes all elements, makes list empty

70

Maps & Multimaps

manage (key, value) pairs

71

overview

- automatically sort elements
- need sort criteria for key
- multimaps allow dupes, maps do not
- implemented as balanced binary trees
- good performance when searching on key
- searching for non-key items may not perform well

72

(key, value) pair

- if *pos* is an iterator pointing to an item in the map:
  - *pos->first* yields the key of the item
  - *pos->second* yields the value

73

three ways to insert (key, value) pair

- in maps and multimaps, the key is constant
- must either provide the correct type or provide implicit or explicit type conversion
- three ways:
  - make_pair() – makes a pair that contains 2 values passed
  - pair()
  - value_type()

74

```
#include <map>

void print(const map<string, int> & my_map) {
   map<string, int>::const_iterator ptr = my_map.begin();
   while ( ptr != my_map.end() ) {
      cout << (*ptr).first << "\t"
           << (*ptr).second << endl;
      ++ptr;
   }
}

int main() {
   map<string, int> my_map;
   pair<string, int> new_entry("boy", 3);
   my_map.insert(new_entry);
   my_map.insert(make_pair(string("girl"), 2));
   my_map.insert(make_pair(string("animal"), 23));
   my_map.insert(make_pair(string("lady"), 12));
   print( my_map );
}
```

75

brackets included

```
#include <map>

int main() {
   map<string, int> my_map;
   pair<string, int> new_entry("boy", 3);
   my_map.insert(new_entry);
   my_map.insert(make_pair(string("girl"), 2));
   my_map.insert(make_pair(string("animal"), 23));
   my_map.insert(make_pair(string("lady"), 12));
   cout << my_map["girl"] << endl;
```

76

key that's not a string

```
#include <map>

class Test {
public:
  Test(int n) : number(n) {}
  int get() const { return number; }
private:
  int number;
};

int main() {
   map<Test *, int> my_map;
   Test * t = new Test(8);
   pair<Test *, int> new_entry(t, 3);
   my_map.insert(new_entry);
   my_map.insert(make_pair(new Test(99), 2));
   map<Test *, int>::const_iterator ptr = my_map.find(t);
   if ( ptr != my_map.end() ) cout << ptr->second << endl;
   print( my_map );
}
```

77

C++ Strings

easier and safer
than C-Strings

78

Functionality for string s:

- All constructors
- s.length(), s.size()
- cin >> s and getline(cin, s)
  - these are very different!
- Concatenation, relops all defined

79

I/O for strings

istream &
getline(istream * input, string str, char delimeter = '\n');

Note the difference between the following:

```
string line;
fstream input;
input >> line;
getline(input, line);
```

80

Sub strings

There are lots of search functions for strings in the library

find() is the most straightforward. Given a string:
-- it returns the index of the first character of the matching substring, or
-- it returns special value string::npos

```
string name("AnnaBelle");
int pos = name.find("Anna");
if  (pos == string::npos) cout << "not found"
else cout << "found";
```

the return type of find is int or string::size_type

81

find_first_of()

```
string numerics("0123456789");
string name("r2d2");

string::size_type pos = name.find_first_of(numerics);
if  (pos == string::npos) cout << "not found"
else cout << "found numeric at: "
          << pos  << endl;
```

there are two versions of find_first_of

82

find_first_of

```
string numerics("0123456789");
string name("r2d2");

string::size_type pos = 0;
while ( (pos = name.find_first_of(numerics, pos) ) != npos)  {
        cout << "numeric found at: " << pos << endl;
        ++pos;
}
```

83

substrings

```
substr(start, how_many);
```

generates a copy of a substring for an existing string object. The first argument specifies where to start and the optional second argument specifies how many to copy. If the 2nd argument is omitted, the rest of the string is copied.

```
string  a("catalog");
cout << a.substr(0, 3) << endl;
```

84

insertion into a string

```
insert(position, string);

string state("Missisippi");
string::size_type pos = state.find("isi");
state.insert(pos+1, "s");

string apple("apple");
apple.insert(apple.size(), "s");
```


erasing substrings

several forms:

```
erase(position);
erase(start, how_many);
```


Exercises for string:

- Write an iterator on string
- write a program that counts the number of vowels in a sentence
- read a sequence of words and print them in alphabetical order
- Write a program that reads a paragraph and prints all the unique words.
- Search the web for info about standard library strings

Exercise for string

- Write a program to play hangman. Your program should pick a word and display:

  Guess the word: XXXXX

  Each X represents a letter. Fill the letters in for correct guesses, for incorrect guesses, build

```
  O
/ | \
  |
/   \
```

Web resources:

- www.josuttis.com/libbook/
- http://www.sgi.com/cgi-bin/surfsearch2.cgi
- http://www.sgi.com/Technology/STL/other_resources.html
- http://www.cyberport.com/~tangent/programming/stl/compatibility.html

89

some examples of using vectors

random access, iterator access, sorting, searching

90

```
#include <iostream>
#include <vector>
const int MAX = 1024;

void print(const vector<int> & n) {
     for (int i = 0; i < n.size(); ++i) cout << n[i] << endl;
}
int main() {
   vector<int> items;
   cout << "size of items is: " << items.size() << endl;
   items.push_back(rand() % 100);
   cout << "size of items is: " << items.size() << endl;
   for (int i = 0; i < MAX; ++i)
        items.push_back(rand() %100);
   cout << "size of items is: " << items.size() << endl;
   print(items);
}
```

looks like an array

91

```
#include <iostream>
#include <vector>
const int MAX = 1024;

void print(const vector<int> & n) {
     vector<int>::const_iterator ptr = n.begin();
     while (ptr != n.end() ) {
          cout << *ptr << endl;
          ++ptr;
     }
}

void fill(vector<int> & n) {
   for  (int i = 0; i < MAX; ++i)   n.push_back(rand()%100);
}

int main() {
   vector<int> items;
   fill(items);   fill(items);
   cout << "size of items is: " << items.size() << endl;
   print(items);
}
```

compatible with other containers

92

```
#include <iostream>
#include <vector>
const int MAX = 1024;

void print(const vector<int> & n) {
     vector<int>::const_iterator ptr = n.begin();
    while ( ptr != n.end() )    cout << *ptr++ << endl;
}

void fill(vector<int> & n) {
  for  (int i = 0; i < MAX; ++i)   n.push_back(rand()%100);
}

int main() {
  vector<int> items;
  fill(items);   fill(items);
  cout << "size of items is: " << items.size() << endl;
  print(items);
}
```

Don't try this at home ;-)

93

```
#include <iostream>
#include <vector>
const int MAX = 1024;

void print(const vector<int> & n) {
     vector<int>::const_iterator ptr = n.begin();
     while ( ptr != n.end() )    cout << *ptr++ << endl;
}

void fill(vector<int> & n) {
  for  (int i = 0; i < MAX; ++i)   n.push_back(rand()%100);
}

int main() {
  vector<int> items;
  fill(items);
  vector<int>::iterator ptr = n.begin();
  ++ptr;
   items.erase(ptr);
  cout << "size of items is: " << items.size() << endl;
  print(items);
}
```

Need an iterator to erase an item

what item was eliminated?
how many items remain?

94

```
// can switch to list and it still works!
#include <iostream>
#include <list>
const int MAX = 1024;

void print(const list<int> & n) {
     list<int>::const_iterator ptr = n.begin();
     while (ptr != n.end() )  cout << *ptr++ << endl;
}
void fill(list<int> & n) {
  for  (int i = 0; i < MAX; ++i)   n.push_back(rand()%100);
}

int main() {
  list<int> items;
  fill(items);   fill(items);
  cout << "size of items is: " << items.size() << endl;
  print(items);
}
```

We switched to list!

95

```
#include <iostream>
#include <vector>
#include <algorithm>
const int MAX = 10;

void print(const vector<int> & n) {
     vector<int>::const_iterator ptr = n.begin();
     while ( ptr != n.end() )    cout << *ptr++ << endl;
}

void fill(vector<int> & n) {
  for  (int i = 0; i < MAX; ++i)   n.push_back(rand()%100);
}

int main() {
  vector<int> items;
  fill(items);
  sort( items.begin(), items.end() );
  print(items);
}
```

it's easy to sort

96

```
#include <iostream>
#include <vector>
#include <algorithm>
const int MAX = 10;

void fill(vector<int> & n) {
  for  (int i = 0; i < MAX; ++i)   n.push_back(rand()%100);
}

void locate(const vector<int> & v, int x) {
  vector<int>::const_iterator ptr = find( v.begin(), v.end(), x );
  if (ptr == v.end() ) cout << x << " not found" << endl;
  else cout << x << " found at location: " << ptr-v.begin() << endl;
}

int main() {
  vector<int> items;
  fill(items);
  locate(items, 99);
  print(items);
}
```

also easy to search

note: we're using search in algorithm

97

```
#include <iostream>
#include <vector>
#include <algorithm>
const int MAX = 10;

void fill(vector<int> & n) {
  for  (int i = 0; i < MAX; ++i)   n.push_back(rand()%100);
}

void locate(const vector<int> & v, int x) {
  if ( binary_search( v.begin(), v.end(), x ) )  cout << x << " found\n";
  else cout << x << " not found" << endl;
}

int main() {
  vector<int> items;
  fill(items);
  sort( items.begin(), items.end() );
  locate(items, 99);
  print(items);
}
```

binary search should be faster than find

container must be sorted

98

vector: inserting & removing

c.insert(pos, elem);	inserts at iterator position *pos*, a copy of *elem* and returns the position of the new element
c.push_back(elem);	appends a copy of *elem* at end
c.pop_back();	removes last element (void fn)
c.erase(pos);	removes the element at iterator position *pos* and returns the position of the next element
c.clear();	removes all elements, makes vector empty

99

vector: inserting & removing

c.resize(num);	changes the number of elements to num, if size grows uses the default construtor to init new locations
c.resize(num, elem);	changes the number of elements to num, if size grows copies elem into new locations

There are lots more: see a good reference, like Josuttis

100

Ordinary use:

```
const int MAX = 5;
int main() {
   vector<int> vec;
   cout << vec.size() << '\t' << vec.capacity() << endl;
   for (int I = 0; I < MAX; ++I; {
      vec.push_back(I);
      cout << vec.size() << '\t' << vec.capacity() << endl;
   }
}
```

101

use reserve():

```
const int MAX = 5;
int main() {
   vector<int> vec;
   vec.reserve(MAX);
   cout << vec.size() << '\t' << vec.capacity() << endl;
   for (int I = 0; I < MAX; ++I; {
      vec.push_back(I);
      cout << vec.size() << '\t' << vec.capacity() << endl;
   }
}
```

102

use constructor:

```
const int MAX = 5;
int main() {
   vector<int> vec(MAX);
   cout << vec.size() << '\t' << vec.capacity() << endl;
   for (int I = 0; I < MAX; ++I; {
      vec.push_back(I);
      cout << vec.size() << '\t' << vec.capacity() << endl;
   }
}
```

103

insert()/erase():

```
const int MAX = 5;
int main() {
   vector<int> vec(MAX);
   cout << vec.size() << '\t' << vec.capacity() << endl;
   for (int I = 0; I < MAX; ++I; {
      vec.push_back(I);
      cout << vec.size() << '\t' << vec.capacity() << endl;
   }
   vec.insert(vec.begin(), 99);
}
```

104

algorithms

sorting and searching
sequential containers

sort of sequential containers can be easy

```
#include <vector>
#include <iostream>
#include <algorithm>

int main() {
  vector<int> vec;
  for (int I = 0; I < MAX; ++I) vec.push_back(rand() % 100);
  sort(vec.begin(), vec.end());
}
```

sort works for any item that has **<** defined for it.


```
#include <vector>
#include <algorithm>
class Student {
public:
  string getName() const { return name; }
  bool operator<(const Student & rhs) const {
    return name < rhs.getName();
  }
private:
  string name;
};

int main() {
  vector<Student> items;
  sort(items.begin(), items.end() );
}
```


Sorting list

- A list is not a random access container
- must use sort routine in list

```
#include <iostream>
#include <list>
const int MAX = 1024;

void print(const list<int> & n) {
     list<int>::const_iterator ptr = n.begin();
     while (ptr != n.end() )  cout << *ptr++ << endl;
}
void fill(list<int> & n) {
   for  (int i = 0; i < MAX; ++i)   n.push_back(rand()%100);
}

int main() {
   list<int> items;
   fill(items);   fill(items);
   items.sort();
   print(items);
}
```


find can be easy

```
#include <vector>
#include <iostream>
#include <algorithm>

int main() {
   vector<int> vec;
   for (int I = 0; I < MAX; ++I) vec.push_back(rand() % 100);
   vector<int>::const_iterator ptr =
         find(vec.begin(), vec.end(), 99);
}
```

find works for any item that has **==** defined for it.