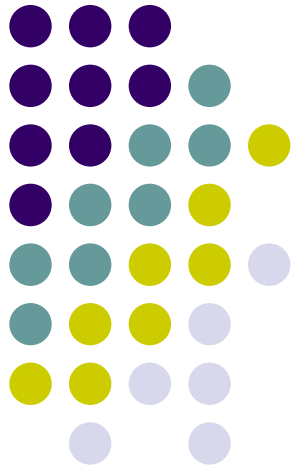
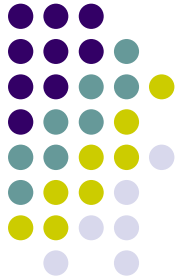
Standard Template Library (STL)

Xiaoqiang Wang

Florida State University

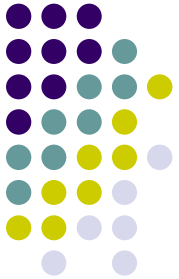


History



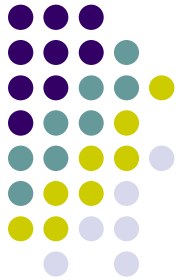
- Alex Stepanov & Meng Lee, based on earlier work by Stepanov & Musser
- Proposed to C++ committee late ‘93
- HP version released ‘94
- Accepted into Standard summer ‘94
- Standard froze ‘97, ratified ‘98

Advantages



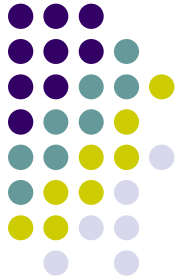
- Standardized
- Thin & efficient
- Little inheritance; no virtual functions
- Small; easy to learn
- Flexible and extensible
- Naturally open source

Standard Template Library



- The standard template library (STL) contains
  - Containers
  - Algorithms
  - Iterators
- A *container* is a way that stored data is organized in memory, for example an array of elements.
- *Algorithms* in the STL are procedures that are applied to containers to process their data, for example search for an element in an array, or sort an array.
- *Iterators* are a generalization of the concept of pointers, they point to elements in a container, for example you can increment an iterator to point to the next element in an array

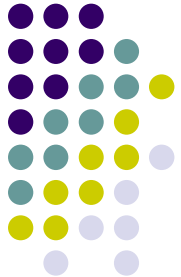
Putting the STL into Action



- Include files have no “.h”
- Standard namespace

```
#include <cstdio>  // new include method
#include <vector>  // vector container
#include <algorithm>      // STL algorithms
using namespace std;      // assume std::

vector<int> Chuck; // declare a growable array
Chuck.push_back(1);       // add an element
find(Chuck.begin(), Chuck.end(), 1);
```

Containers



- Containers *contain* elements; they “own” the objects
- Containers provide iterators that point to its elements.
- Containers provide a minimal set of operations for manipulating elements

Containers (cont.)

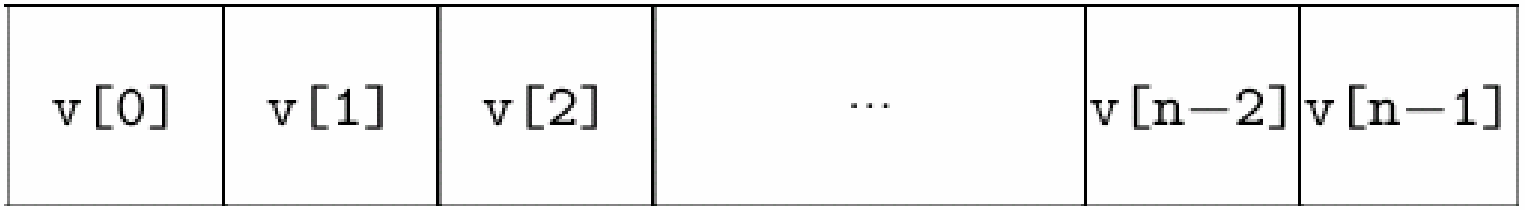
Container	Description	Keys
vector	dynamic array	
deque	dynamic array -- both ends	
list	linked list	
set	sorted list of keys	no duplicate keys
map	sorted list of key and value pairs	no duplicate keys
multiset	sorted list of keys	duplicate keys OK
multimap	sorted list of key and value pairs	duplicate keys OK

- Minimum container object requirements

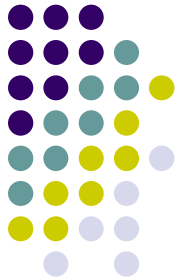
```
X()                                 // default ctor
X(const X&)                         // copy ctor
X& operator = (const X&)            // assignment op
bool operator < (const X&)          // comparison op
bool operator == (const X&)         // comparison op
```

Vectors

- Generalization of arrays
- Efficient, random-access to elements
- High-level operations such as increasing/decreasing size of vector



Strings

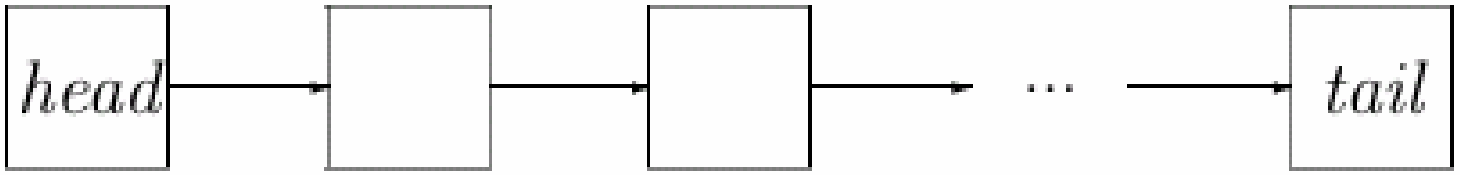


- In one sense, a vector of characters
- In another sense, a completely different high-level data type
- Lots of string-specific operations (more later)

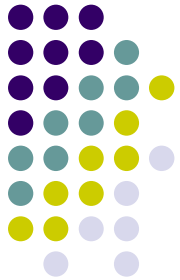
```
string aName = "Benjamin Franklin";
```

Lists

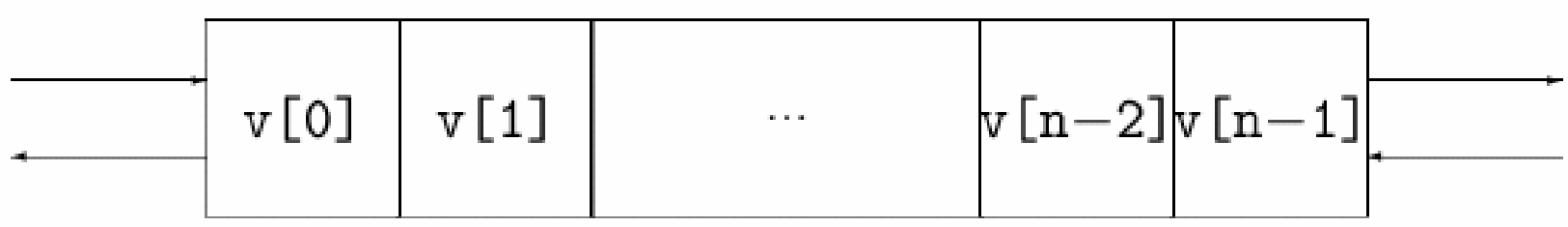
- Arbitrary size, memory used efficiently as the list grows and shrinks
- Sequential access only, constant access to first or last element
- Efficient insertion or removal at any location



Deque – Double Ended Queue

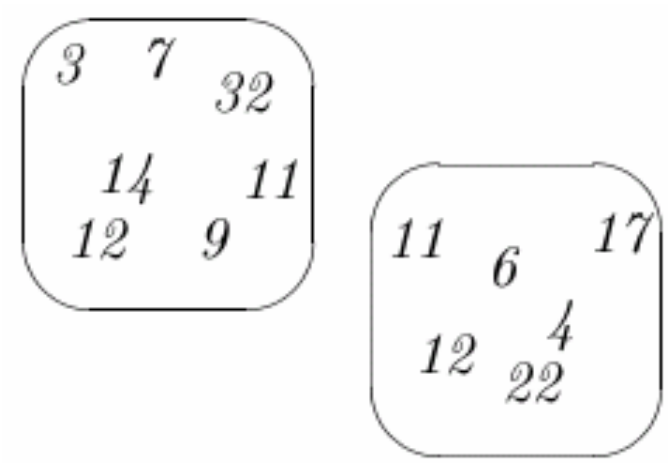


- Grows or shrinks as necessary
- Efficient insertion or removal from either end
- Random access to elements

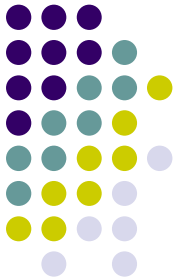


Sets

- Ordered collection
- Efficient (logarithmic) insertion, removal, and test for inclusion
- Efficient merge, union, difference, and other set operations
- Multiset allows more than one entry with the same value



Map (Dictionary)



- Collection of (key, value) pairs
- Keys can be any ordered data type (e.g. string)
- Values can be any data type
- Efficient insertion, removal, and test for inclusion

$$
\begin{aligned}
key_1 &\rightarrow value_1 \\
key_2 &\rightarrow value_2 \\
key_3 &\rightarrow value_3 \\
&\ldots \\
key_n &\rightarrow value_n
\end{aligned}
$$

Container Adaptors

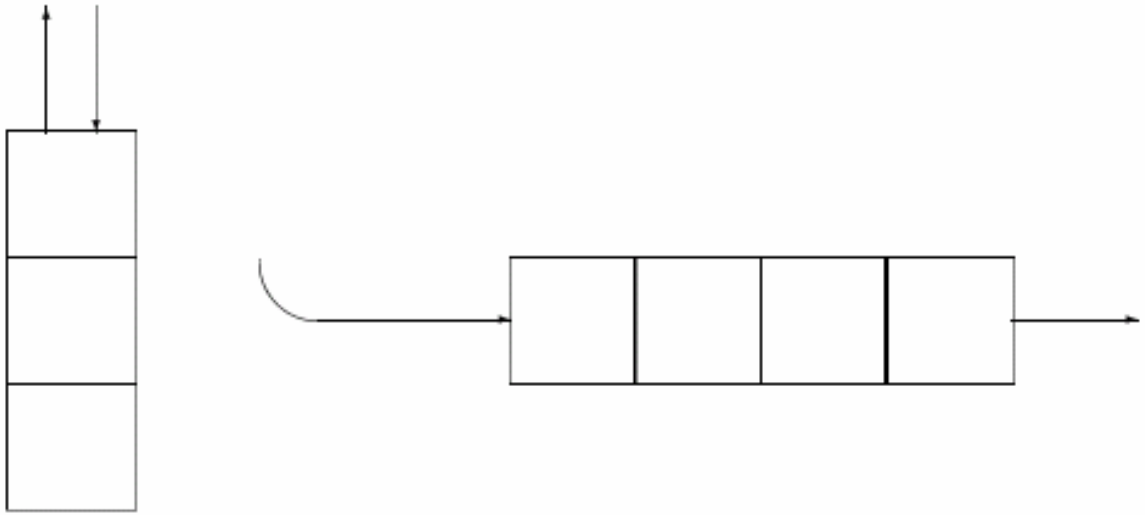
Adaptor	Example containers	Default container
stack	list, deque, vector	deque
queue	list, deque	deque
priority_queue	list, deque, vector	vector

- Example adapator code

```
stack<int, deque<int> > TechnoTrousers;
TechnoTrousers.push(1);
int i = TechnoTrousers.top();
TechnoTrousers.pop();
```

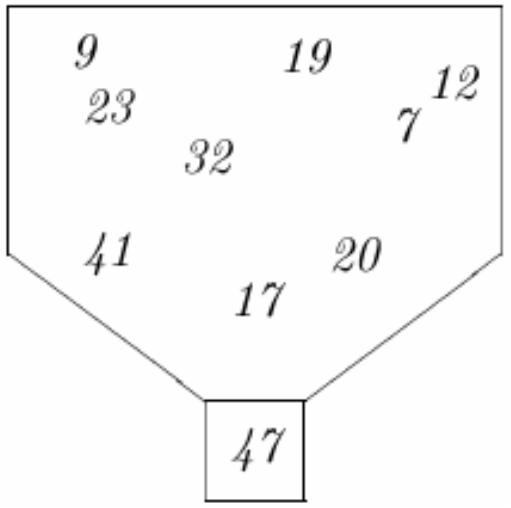
Stacks and Queues

- Special form of deque
- Stack = Last in First out (LIFO)
- Queue = First in First out (FIFO)

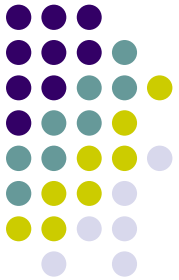


Priority Queue

- Efficient (logarithmic) insertion of new values
- Efficient access to largest (or smallest) value
  - Constant time access
  - Logarithmic removal



What′s Missing?



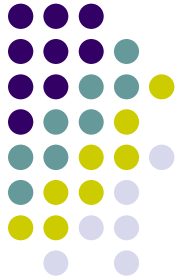
- stack-based arrays (T a[N])
- hash tables
- singly-linked lists
- some STL implementations include one or more of these “non-standard” containers

Container Efficiency

Container	Overhead	Insert	Erase	[]	Find	Sort
list	8	C	C	n/a	N	N log N
deque	12	C at begin or end; else N/2	C at begin or end; else N	C	N	N log N
vector	0	C at end; else N	C at end; else N	C	N	N log N
set	12	log N	log N	n/a	log N	C
multiset	12	log N	d log (N+d)	n/a	log N	C
map	16	log N	log N	log N	log N	C
multimap	16	log N	d log (N+d)	log N	log N	C
stack	n/a	C	C	n/a	n/a	n/a
queue	n/a	C	C	n/a	n/a	n/a
priority_ queue	n/a	log N	log N	n/a	n/a	n/a
slist (SGI)	4	C	C	n/a	N	n/a
hashset (SGI)	?	C/N	C/N	n/a	C/N	n/a
hashmap (SGI)	?	C/N	C/N	n/a	C/N	n/a

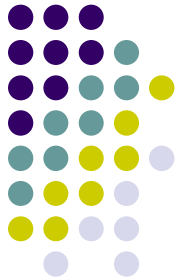
- Overhead is approx. per-element size in bytes
- Hash and slist containers included for comparison only. C/N indicates best/worst case times

Selecting a Container Class



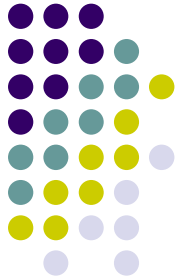
- How are values going to be accessed?
  - Random – vector or deque
  - Ordered – set or map
  - Sequential – list
- Is the order in which values are maintained in the collection important?
  - Ordered – set or map
  - Can be sorted – vector or deque
  - Insertion time dependant – stack or queue

Selecting a Container Class (cont)



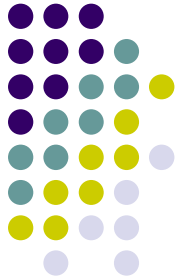
- Will the size of the structure vary widely over the course of execution?
  - Yes – list or set
  - No – vector or deque
- Is it possible to estimate the size of the collection?
  - Yes – vector

Selecting a Container Class (cont)



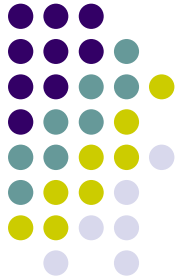
- Is testing to see whether a value is contained in the collection a frequent operation?
  - Yes – set
- Is the collection indexed?
  - Index values are integers – vector or deque
  - Index values are not integers – map

Selecting a Container Class (cont)



- Can values be related to each another?
  - Sets require relational operators
  - Vectors and lists do not require relational operators
- Is finding and removing the largest (or smallest) value from the collection a frequent operation?
  - Yes – priority queue

Selecting a Container Class (cont)

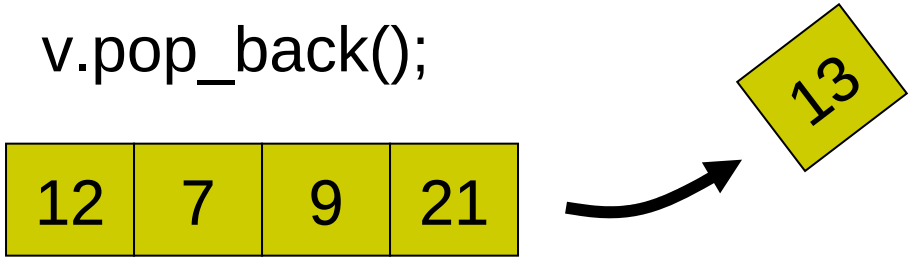


- At what positions are items inserted into and removed from the collection?
  - Middle – list
  - End – stack or queue
- Is a frequent operation the merging of two or more sequences into one?
  - Ordered – set
  - Unordered – list

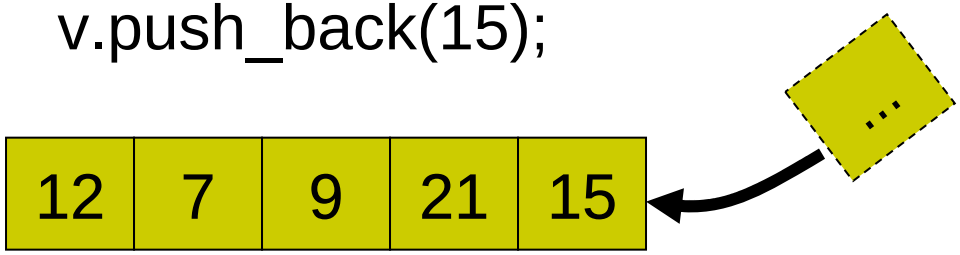
Vector Container

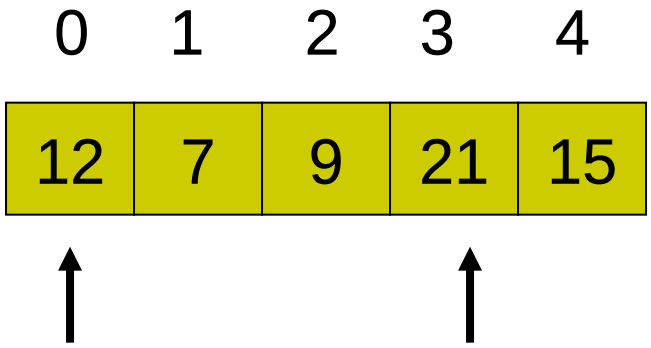
```
int array[5] = {12, 7, 9, 21, 13 };
vector<int> v(array,array+5);
```

v.pop_back();



v.push_back(15);

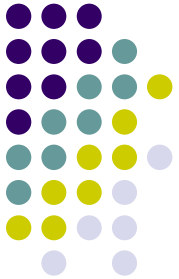




v.begin();

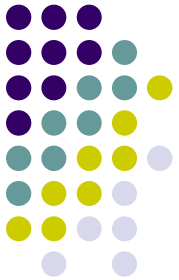
v[3]

Vector Container



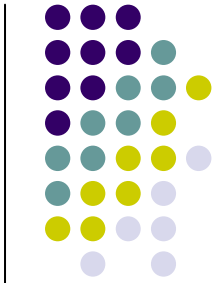
```
#include <vector>
#include <iostream>
vector<int> v(3);  // create a vector of ints of size 3
v[0]=23;
v[1]=12;
v[2]=9;    // vector full
v.push_back(17);   // put a new value at the end of array
for (int i=0; i<v.size(); i++)   // member function size() of vector
   cout << v[i] << " ";   // random access to i-th element
cout << endl;
```

Vector Container



```
#include <vector>
#include <iostream>
int arr[] = { 12, 3, 17, 8 };  // standard C array
vector<int> v(arr, arr+4);  // initialize vector with C array
while ( ! v.empty()) // until vector is empty
{
    cout << v.back() << ” ”;   // output last element of vector
    v.pop_back();              // delete the last element
}
cout << endl;
```

Constructors for Vector

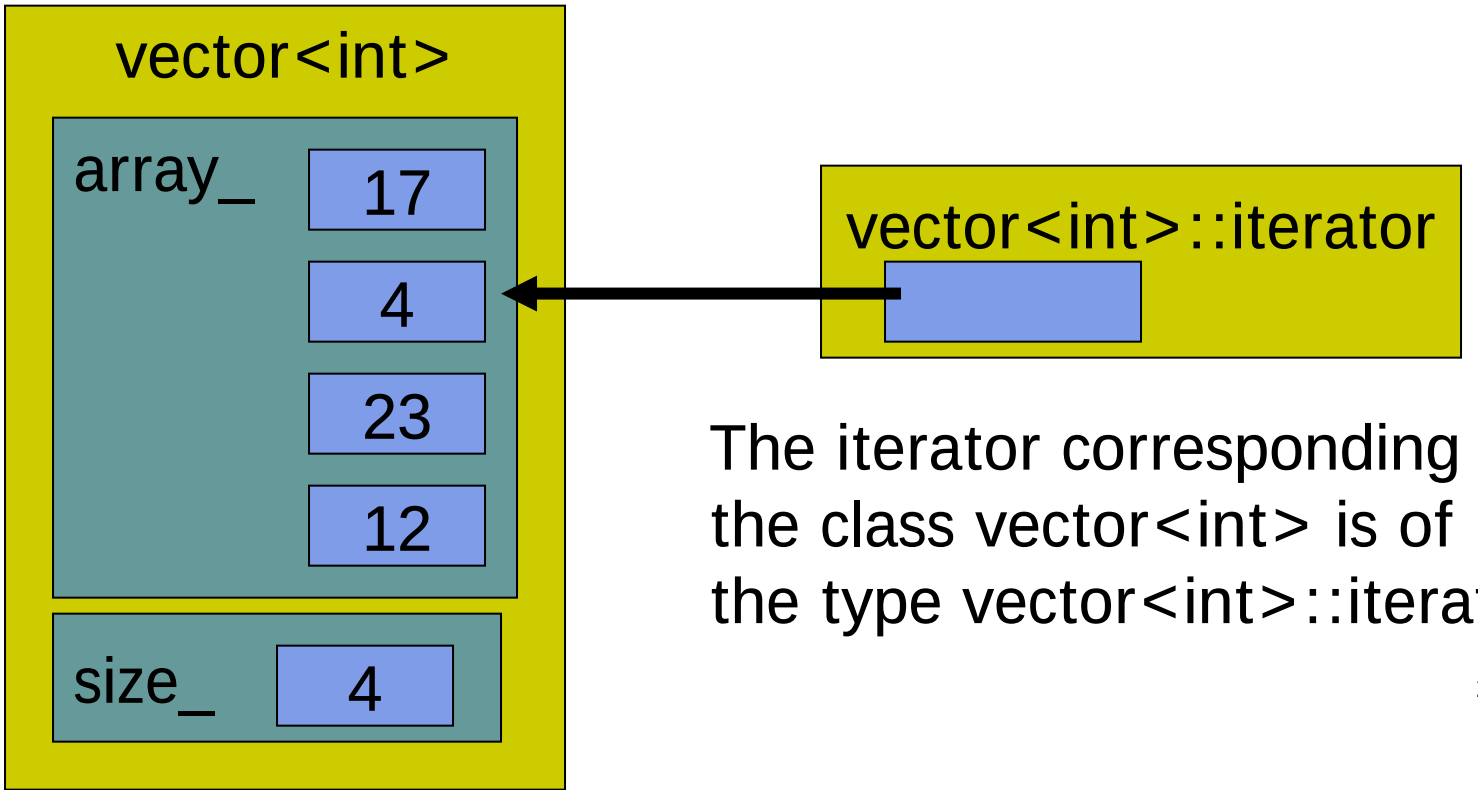


- A vector can be initialized by specifying its size and a prototype element or by another vector

```
vector<Date> x(1000); // creates vector of size 1000,
                      // requires default constructor for Date
vector<Date> dates(10,Date(17,12,1999)); // initializes
                      // all elements with 17.12.1999
vector<Date> y(x); // initializes vector y with  vector x
```

Iterators

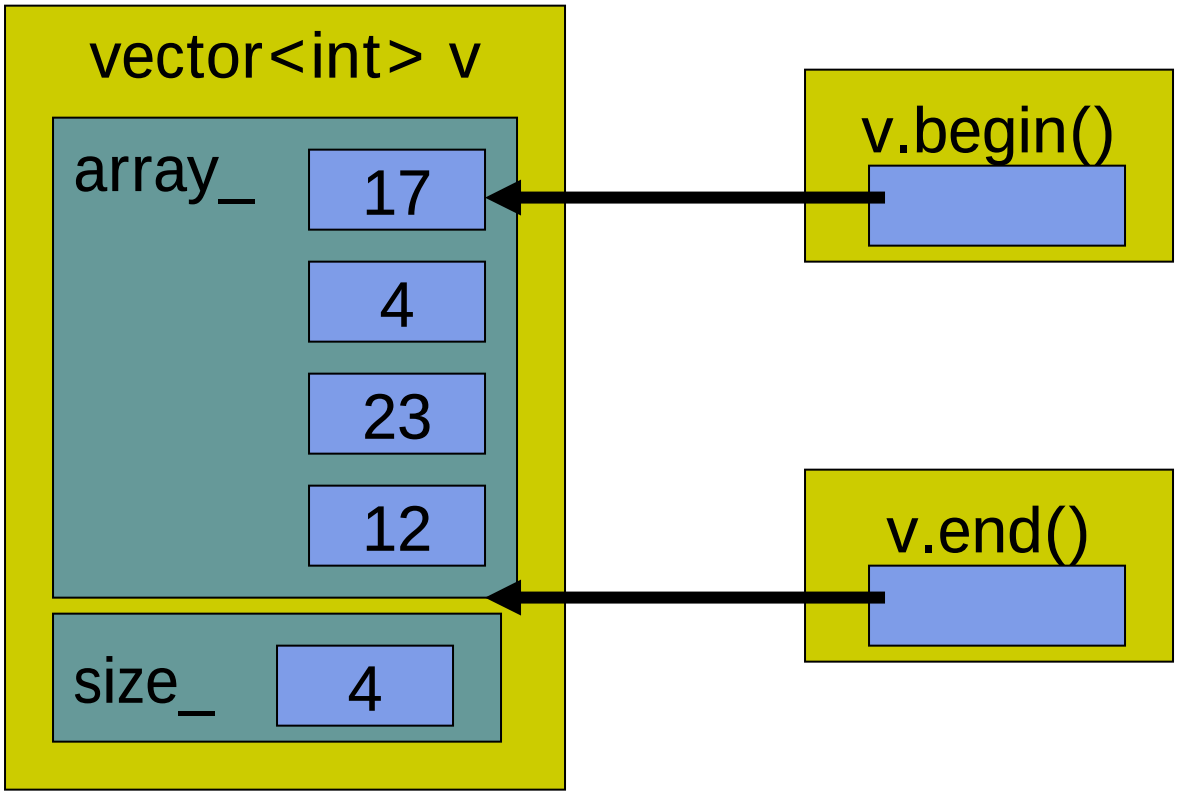
- Iterators are pointer-like entities that are used to access individual elements in a container.
- Often they are used to move sequentially from element to element, a process called *iterating* through a container.



The iterator corresponding to the class vector<int> is of the type vector<int>::iterator

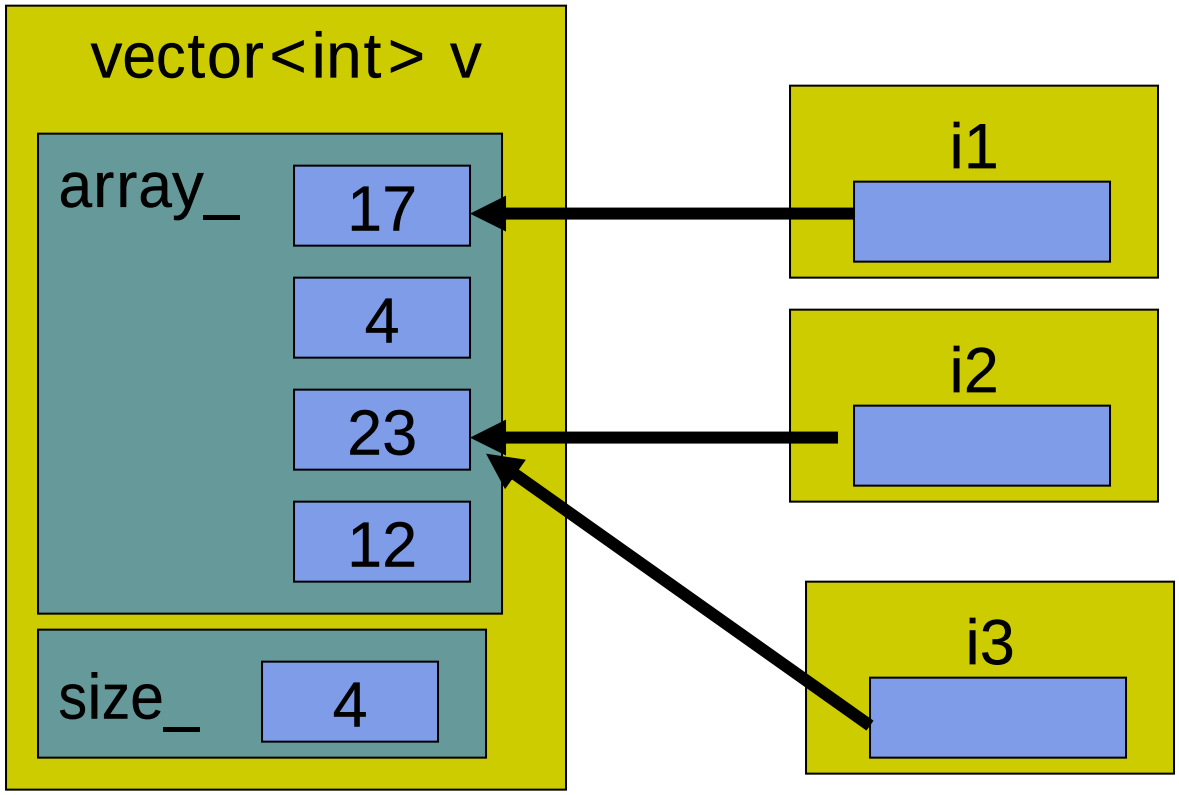
Iterators

- The member functions begin() and end() return an iterator to the first and past the last element of a container

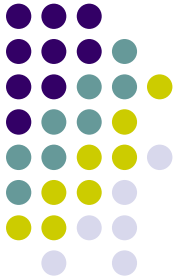


Iterators

- One can have multiple iterators pointing to different or identical elements in the container

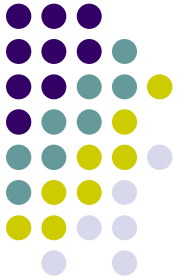


Iterators



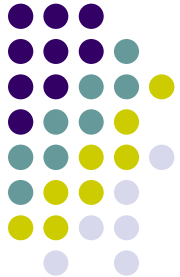
```
#include <vector>
#include <iostream>

int arr[] = { 12, 3, 17, 8 };  // standard C array

vector<int> v(arr, arr+4);  // initialize vector with C array

vector<int>::iterator iter=v.begin();  // iterator for class vector
// define iterator for vector and point it to first element of v

cout << ”first element of v=” << *iter; // de-reference iter

iter++;  // move iterator to next element

iter=v.end()-1;  // move iterator to last element
```

Iterators



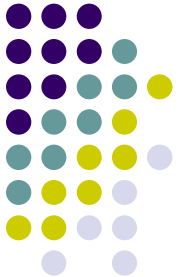
```
int max(vector<int>::iterator start, vector<int>::iterator end)
{
    int m=*start;
    while(start != end)
     {
        if (*start > m)
           m=*start;
        ++start;
     }
     return m;
}
cout << ”max of v = ” << max(v.begin(),v.end());
```

Iterators



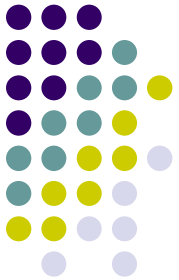
```
#include <vector>
#include <iostream>
int arr[] = { 12, 3, 17, 8 };  // standard C array
vector<int> v(arr, arr+4);  // initialize vector with C array

for (vector<int>::iterator i=v.begin(); i!=v.end(); i++)
// initialize i with pointer to first element of v
// i++ increment iterator, move iterator to next element
{
    cout << *i << " ";   // de-referencing iterator returns the
                         // value of the element the iterator points at
}
cout << endl;
```

For_Each() Algorithm



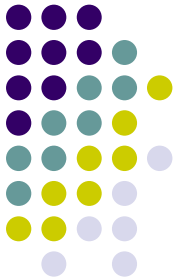
```
#include <vector>
#include <algorithm>
#include <iostream>
void show(int n)
{
  cout << n << ” ”;
}

 int arr[] = { 12, 3, 17, 8 };  // standard C array
vector<int> v(arr, arr+4);  // initialize vector with C array
// apply function show to each element of vector v
for_each (v.begin(), v.end(), show);
```

Find() Algorithm



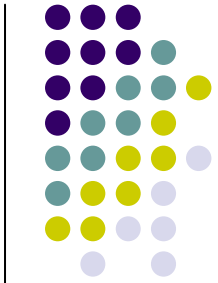
```
#include <vector>
#include <algorithm>
#include <iostream>
int key;
int arr[] = { 12, 3, 17, 8, 34, 56, 9  };  // standard C array
vector<int> v(arr, arr+7);  // initialize vector with C array
vector<int>::iterator iter;
cout << ”enter value :”;
cin >> key;
iter=find(v.begin(),v.end(),key); // finds integer key in v
if (iter != v.end()) // found the element
   cout << ”Element ” << key << ” found” << endl;
else
  cout << ”Element ” << key << ” not in vector v” << endl;
```

Find_If() Algorithm



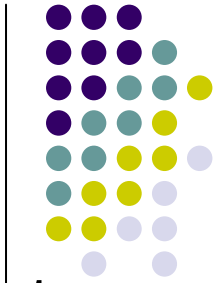
```
#include <vector>
#include <algorithm>
#include <iostream>
Bool mytest(int n) { return (n>21) && (n <36); };
int arr[] = { 12, 3, 17, 8, 34, 56, 9  };  // standard C array
vector<int> v(arr, arr+7);  // initialize vector with C array
vector<int>::iterator iter;

iter=find_if(v.begin(),v.end(),mytest);
  // finds element in v  for which mytest is true

if (iter != v.end()) // found the element
   cout << ”found ” << *iter << endl;
else
  cout << ”not found” << endl;
```

Count_If() Algorithm



```
#include <vector>
#include <algorithm>
#include <iostream>
Bool mytest(int n) { return (n>14) && (n <36); };

int arr[] = { 12, 3, 17, 8, 34, 56, 9  };  // standard C array
vector<int> v(arr, arr+7);  // initialize vector with C array

int n=count_if(v.begin(),v.end(),mytest);
  // counts element in v  for which mytest is true

cout << ”found ” << n << ” elements” << endl;
```

List Container

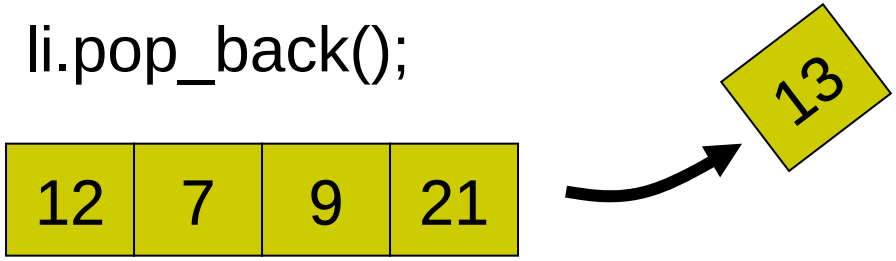


- An STL list container is a double linked list, in which each element contains a pointer to its successor and predecessor.
- It is possible to add and remove elements from both ends of the list
- Lists do not allow random access but are efficient to insert new elements and to sort and merge lists

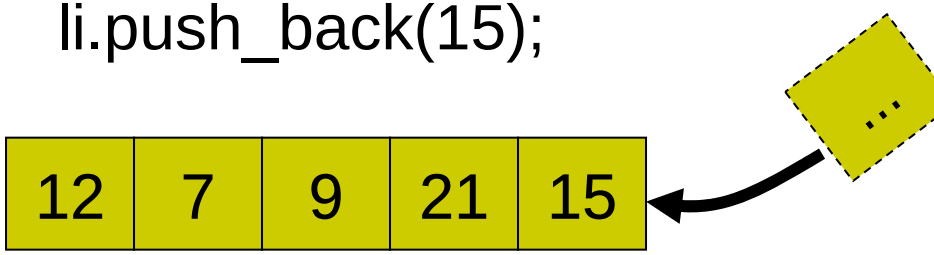
List Container

```
int array[5] = {12, 7, 9, 21, 13 };
list<int> li(array,array+5);
```

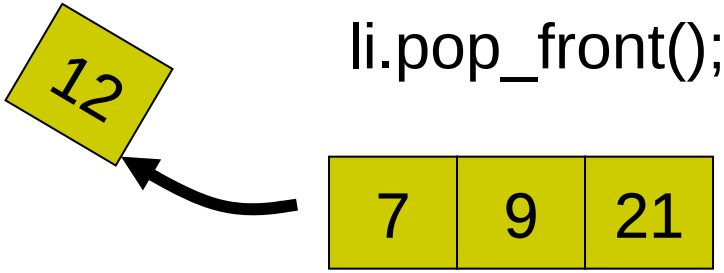
li.pop_back();



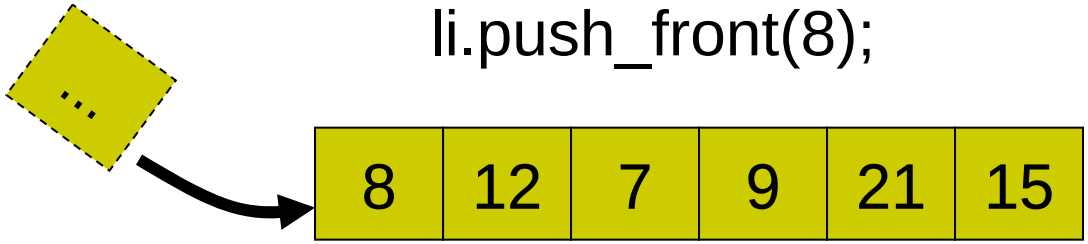
li.push_back(15);



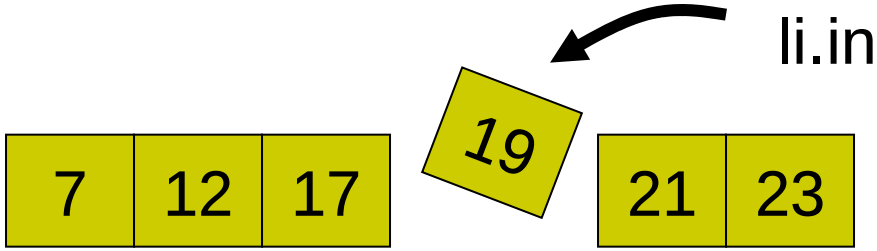
li.pop_front();



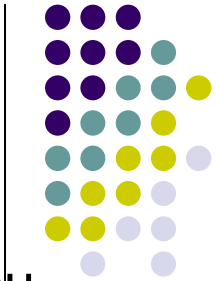
li.push_front(8);



li.insert()



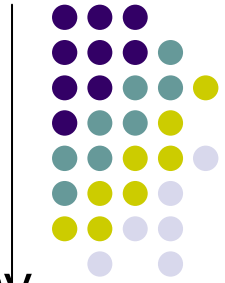
Insert Iterators



- If you normally copy elements using the copy algorithm you overwrite the existing contents

```
#include <list>
int arr1[]= { 1, 3, 5, 7, 9 };
int arr2[]= { 2, 4, 6, 8, 10 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
list<int>  l2(arr2, arr2+5); // initialize l2 with arr2
copy(l1.begin(), l1.end(), l2.begin());
   // copy contents of l1 to l2 overwriting the elements in l2
   // l2 = { 1, 3, 5, 7, 9 }
```

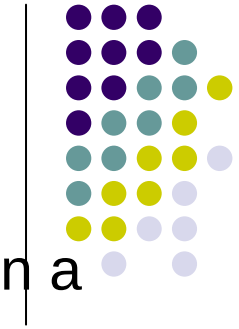
Insert Iterators



- With insert operators you can modify the behavior of the copy algorithm
  - back_inserter : inserts new elements at the end
  - front_inserter : inserts new elements at the beginning
  - inserter : inserts new elements at a specified location

```
#include <list>
int arr1[]= { 1, 3, 5, 7, 9 };
int arr2[]= { 2, 4, 6, 8, 10 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
list<int>  l2(arr2, arr2+5); // initialize l2 with arr2
copy(l1.begin(), l1.end(), back_inserter(l2));  // use back_inserter
   // adds contents of l1 to the end of l2 = { 2, 4, 6, 8, 10, 1, 3, 5, 7, 9  }
copy(l1.begin(), l1.end(), front_inserter(l2));  // use front_inserter
   // adds contents of l1 to the front of l2 = { 9, 7, 5, 3, 1, 2, 4, 6, 8, 10 }
copy(l1.begin(), l1.end, inserter(l2,l2.begin());
 // adds contents of l1 at the ”old” beginning of l2 = { 1, 3, 5, 7, 9, 2, 4, 6, 8, 10 }
```

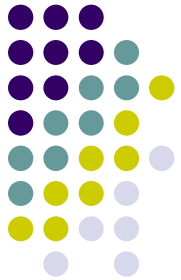
Sort & Merge



- Sort and merge allow you to sort and merge elements in a container

```
#include <list>
int arr1[]= { 6, 4, 9, 1, 7 };
int arr2[]= { 4, 2, 1, 3, 8 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
list<int>  l2(arr2, arr2+5); // initialize l2 with arr2
l1.sort();  // l1 = {1, 4, 6, 7, 9}
l2.sort(); // l2= {1, 2, 3, 4, 8 }
l1.merge(l2);  // merges l2 into l1
// l1 = { 1, 1, 2, 3, 4, 4, 6, 7, 8, 9},  l2= {}
```

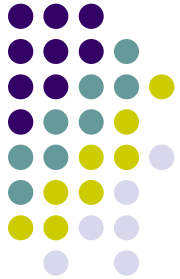
Functions Objects



- Some algorithms like sort, merge, accumulate can take a function object as argument.
- A function object is an object of a template class that has a single member function : the overloaded operator ()
- It is also possible to use user-written functions in place of pre-defined function objects

```
#include <list>
#include <functional>
int arr1[]= { 6, 4, 9, 1, 7 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
l1.sort(greater<int>());  // uses function object greater<int>
// for sorting in reverse order l1 = { 9, 7, 6, 4, 1 }
```

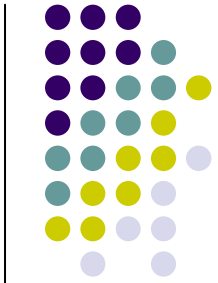
Function Objects



- The accumulate algorithm accumulates data over the elements of the containing, for example computing the sum of elements

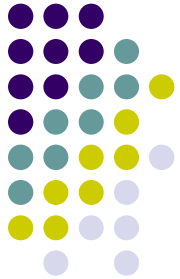
```
#include <list>

#include <functional>

#include <numeric>

int arr1[]= { 6, 4, 9, 1, 7 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1

int sum = accumulate(l1.begin(), l1.end() , 0, plus<int>());

int sum = accumulate(l1.begin(), l1.end(),0);  // equivalent

int fac = accumulate(l1.begin(), l1.end() , 0, times<int>());
```

User Defined Function Objects



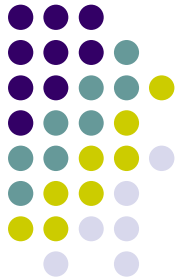
```
class squared _sum  // user-defined function object
{
   public:
      int operator()(int n1, int n2) { return n1+n2*n2; }
};
int sq = accumulate(l1.begin(), l1.end() , 0, squared_sum() );
// computes the sum of squares
```

User Defined Function Objects



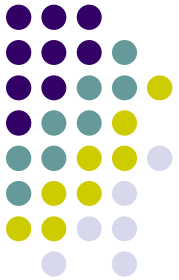
```
template <class T>
class squared _sum  // user-defined function object
{
   public:
      T operator()(T n1, T n2) { return n1+n2*n2; }
};
vector<complex> vc;
complex sum_vc;
vc.push_back(complex(2,3));
vc.push_back(complex(1,5));
vc.push_back(complex(-2,4));
sum_vc = accumulate(vc.begin(), vc.end() ,
                    complex(0,0) , squared_sum<complex>() );
// computes the sum of squares of a vector of complex numbers
```

Associative Containers



- In an associative container the items are not arranged in sequence, but usually as a tree structure or a hash table.
- The main advantage of associative containers is the speed of searching (binary search like in a dictionary)
- Searching is done using a *key* which is usually a single value like a number or string
- The *value* is an attribute of the objects in the container
- The STL contains two basic associative containers
  - sets and multisets
  - maps and multimaps

Sets and Multisets

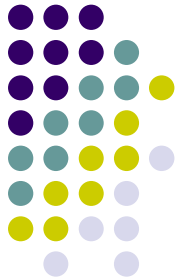


```
#include <set>
string names[] = {”Ole”, ”Hedvig”, ”Juan”, ”Lars”, ”Guido”};
set<string, less<string> > nameSet(names,names+5);
// create a set of names in which elements are alphabetically
// ordered string is the key and the object itself
nameSet.insert(”Patric”); // inserts more names
nameSet.insert(”Maria”);
nameSet.erase(”Juan”); // removes an element
set<string, less<string> >::iterator iter; // set iterator
string searchname;
cin >> searchname;
iter=nameSet.find(searchname);  // find matching name in set
if (iter == nameSet.end())    // check if iterator points to end of set
   cout << searchname << ” not in set!” <<endl;
else
  cout << searchname << ” is in set!” <<endl;
```

Set and Multisets

```
string names[] = {”Ole”, ”Hedvig”, ”Juan”, ”Lars”, ”Guido”, ”Patric”,
    ”Maria”, ”Ann”};
set<string, less<string> > nameSet(names,names+7);
set<string, less<string> >::iterator iter; // set iterator
iter=nameSet.lower_bound(”K”);
// set iterator to lower start value ”K”
while (iter != nameSet.upper_bound(”Q”))
  cout << *iter++ << endl;

// displays Lars, Maria, Ole, Patric
```

Maps and Multimaps

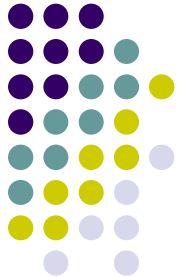


- A map stores pairs <key, value> of a key object and associated value object.
- The key object contains a key that will be searched for and the value object contains additional data
- The key could be a string, for example the name of a person and the value could be a number, for example the telephone number of a person

Maps and Multimaps

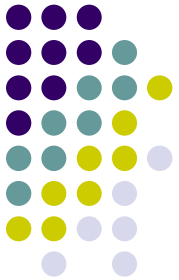
```
#include <map>
string names[]= {”Ole”, ”Hedvig”, ”Juan”, ”Lars”, ”Guido”, ”Patric”, ”Maria”,
    ”Ann”};
int numbers[]= {75643, 83268, 97353, 87353, 19988, 76455, 77443,12221};
map<string, int, less<string> > phonebook;
map<string, int, less<string> >::iterator iter;
for (int j=0; j<8; j++)
   phonebook[names[j]]=numbers[j];  // initialize map phonebook
for (iter = phonebook.begin(); iter !=phonebook.end(); iter++)
   cout << (*iter).first << ” : ” << (*iter).second << endl;
cout << ”Lars phone number is ” << phonebook[”Lars”] << endl;
```

Container Adapters



- container adapters: **`stack`**, **`queue`**, **`priority_queue`**
  - not first class containers
  - do not support iterators
  - do not provide actual data structure
  - programmer can select implementation of the container adapters
  - have member functions **`push()`** and **`pop()`**

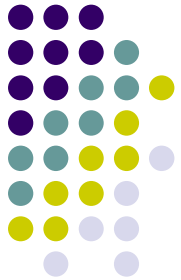
`stack` Adapter



- **`stack`**
  - insertions and deletions at one end
  - last-in-first-out data structure
  - implemented with **`vector`**, **`list`**, and **`deque`** (default)
  - **`#include <stack>`**
- Declarations

```
stack<type, vector<type> > myStack;
stack<type, list<type> > myOtherStack;
stack<type> anotherStack;
```

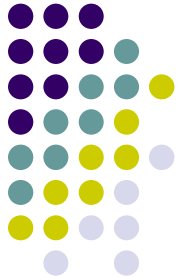
queue Adapter



- **`queue`** - insertions at back, deletions at front
  - first-in-first-out data structure
  - implemented with **`list`** or **`deque`**
  - **`#include <queue>`**
- Functions
  - **`push(element) - (push_back)`** add to end
  - **`pop(element) - (pop_front)`** remove from front
  - **`empty() -`** test for emptiness
  - **`size() -`** returns number of elements
- Example:

```
queue <double> values;  //create queue
values.push(1.2);       // values: 1.2
values.push(3.4);       // values: 1.2 3.4
values.pop();           // values: 1.2
```

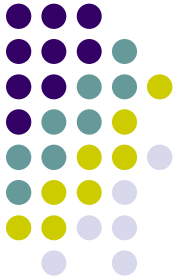
`priority_queue` Adapter



- insertions in sorted order, deletions from front
  - implemented with **`vector`** or **`deque`**
  - highest priority element always removed first
    - heapsort puts largest elements at front
    - **`less<T>`** by default, programmer can specify another

- Functions
  - **`push - (push_back`** then reorder elements**`)`**
  - **`pop - (pop_back`** to remove highest priority element**`)`**
  - **`size`**
  - **`empty`**

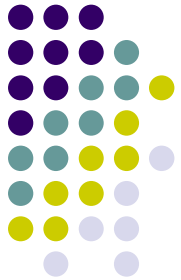
Algorithms



- Before STL
  - class libraries were incompatible among vendors
  - algorithms built into container classes

- STL separates containers and algorithms
  - easier to add new algorithms
  - more efficient, avoids **`virtual`** function calls

for_each, accumulate, transform

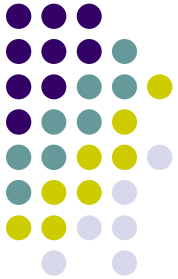


- `for_each(iterator1, iterator2, function);`

- ```
  list<int> L(..);
  int sum = accumulate(L.begin(), L.end(),
                       0, plus<int>());
  ```

- ```
  transform(istream_iterator<Point>(cin),
            istream_iterator<Point>(),
            ostream_iterator<double>(cout),
            mem_fun_ref(Point::x));
  ```

Basic Searching Algorithms



- **`find(iterator1, iterator2, value)`**
  - returns iterator pointing to first instance of value
- **`find_if(iterator1, iterator2, function)`**
  - like **`find,`** but returns an iterator when **`function`** returns **`true`**.
- **`binary_search(iterator1, iterator2, value)`**
  - searches an ascending sorted list for value using a binary search

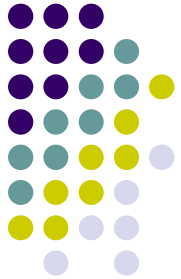
Sorting Algorithms

- **`sort(begin, end)`**
- **`partial_sort(begin, begin+N, end)`**
  - finds first N and sorts them
- **`nth_element(begin, begin+N, end)`**
  - finds first N, not sorted
- **`partition(begin, end, function)`**
  - splits in two intervals

- **`stable_sort, stable_partition`**

- Remarks
  - All take optionaly a comparison function
  - **`std::sort`** is faster than clib sort

`equal`, `mismatch`, `lexicographical_compare`



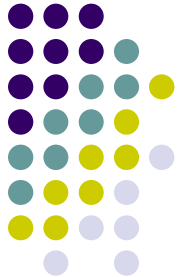
- Functions to compare sequences of values

- **`equal`**
  - returns **`true`** if sequences are equal (uses **`==`**)
  - returns **`false`** if of unequal length

```
equal(iterator1, iterator2, iterator3);
```

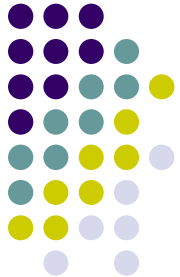
  - compares sequence from **`iterator1`** up to **`iterator2`** with the sequence beginning at **`iterator3`**
  - Containers can be of different types

`fill`, `fill_n`, `generate`, `generate_n`



- STL functions, change containers.
- **`fill -`** changes a range of elements (from **`iterator1`** to **`iterator2`**) to **`value`**
  - **`fill(iterator1, iterator2, value);`**
- **`fill_n`** - changes specified number of elements, starting at **`iterator1`**
  - **`fill_n(iterator1, quantity, value);`**
- **`generate -`** like **`fill`**, but calls a function for value
  - **`generate(iterator1, iterator2, function);`**
- **`generate_n`** - like **`fill_n`**, but calls function for value
  - **`generate_n(iterator1, quantity, function)`**

`swap`, `iter_swap` and `swap_ranges`



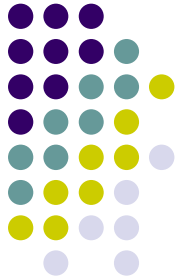
- **`swap(element1, element2)`** - exchanges two values

```
swap( a[ 0 ], a[ 1 ] );
```

- **`iter_swap(iterator1, iterator2) -`** exchanges the values to which the iterators refer

- **`swap_ranges(iterator1, iterator2, iterator3)`** - swap the elements from **`iterator1`** to **`iterator2`** with the elements beginning at **`iterator3`**

copy_backward, merge, unique, reverse



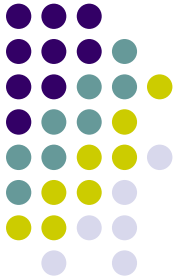
- **copy_backward(iterator1, iterator2, iterator3)**
  - copy the range of elements from **iterator1** to **iterator2** into **iterator3**, but in reverse order.

- **merge(iter1, iter2, iter3, iter4, iter5)**
  - ranges **iter1-iter2** and **iter3-iter4** must be sorted in ascending order.
  - **merge** copies both lists into **iter5**, in ascending order.

- **unique(iter1, iter2) -** removes duplicate elements from a sorted list, returns iterator to new end of sequence.

- **reverse(iter1, iter2)-** reverses elements in the range of **iter1** to **iter2**.

Remove



- Member function:
  ```
  L.remove(Point(0,0));
  ```

- Algorithm moves at end:
  ```
  list<Point> L(istream_iterator<Point>(cin),
                istream_iterator<Point>());

  list<Point>::iterator eit;

  eit = remove(L.begin(), L.end(), Point(0,0));

  L.erase(eit, L.end());
  ```