CHAPTER 19

Standard Template Library

LEARNING OBJECTIVES

- Iterators
  - Constant and mutable iterators
  - Reverse iterators
- Containers
  - Sequential containers
  - Container adapters stack and queue
  - Associative Containers set and map
- Generic Algorithms
  - Big-O notation
  - Sequence, set, and sorting algorithms

INTRODUCTION

- Recall stack and queue data structures
  - We created our own
  - Large collection of standard data structures exists
  - Make sense to have standard portable implementations of them!
- Standard Template Library (STL)
  - Includes libraries for all such data structures
    - Like container classes: stacks and queues

ITERATORS

- Recall: **generalization of a pointer**
  - Typically even implemented with pointer!
- "Abstraction" of iterators
  - Designed to hide details of implementation
  - Provide uniform interface across different container classes
- Each container class has "own" iterator type
  - Similar to how each data type has own pointer type

MANIPULATING ITERATORS

- Recall using overloaded operators:
  - ++, --, ==, !=
  - *
    - So if p is an iterator variable, *p gives access to data pointed to by p
- Vector template class
  - Has all above overloads
  - Also has members begin() and end()

```
c.begin();      //Returns iterator for 1st item in c
c.end();        //Returns "test" value for end
```

CYCLING WITH ITERATORS

- Recall cycling ability:

```
for (p=c.begin();p!=c.end();p++)
        process *p //*p is current data item
```

- Big picture so far…
- Keep in mind:
  - Each container type in STL has own iterator types
    - Even though they're all used similarly

DISPLAY 19.1 ITERATORS USED WITH A VECTOR

```
1   //Program to demonstrate STL iterators.
2   #include <iostream>
3   #include <vector>
4   using std::cout;
5   using std::endl;
6   using std::vector;

7   int main( )
8   {
9       vector<int> container;

10      for (int i = 1; i <= 4; i++)
11          container.push_back(i);

12      cout << "Here is what is in the container:\n";
13      vector<int>::iterator p;
14      for (p = container.begin( ); p != container.end( ); p++)
15          cout << *p << " ";
16      cout << endl;

17      cout << "Setting entries to 0:\n";
18      for (p = container.begin( ); p != container.end( ); p++)
19           *p = 0;
20      cout << "Container now contains:\n";
21      for(p=container.begin();
                          p!=container.end();
                          p++)
22          cout << *p << " ";
23      cout << endl;

24      return 0;
25  }
```

SAMPLE DIALOGUE

Here is what is in the container:
1 2 3 4
Setting entries to 0:
Container now contains:
0 0 0 0

VECTOR ITERATOR TYPES

- Iterators for vectors of ints are of type:

  std::vector<int>::iterator

- Iterators for lists of ints are of type:

  std::list<int>::iterator

- Vector is in std namespace, so need:

```
using std::vector<int>::iterator;
```

KINDS OF ITERATORS

- Different containers → different iterators
- Vector iterators
  - Most "general" form
  - All operations work with vector iterators
  - Vector container great for iterator examples

RANDOM ACCESS: DISPLAY 19.2 BIDIRECTIONAL AND RANDOM-ACCESS ITERATOR USE

```
 7  int main( )
 8  {
 9      vector<char> container;

10      container.push_back('A');
11      container.push_back('B');
12      container.push_back('C');
13      container.push_back('D');

14      for (int i = 0; i < 4; i++)
15          cout << "container[" << i << "] == "
16               << container[i] << endl;

17      vector<char>::iterator p = container.begin( );
18      cout << "The third entry is " << container[2] << endl;
19      cout << "The third entry is " << p[2] << endl;
20      cout << "The third entry is " << *(p + 2) << endl;

21      cout << "Back to container[0].\n";
22      p = container.begin( );
23      cout << "which has value " << *p << endl;

24      cout << "Two steps forward and one step back:\n";
25      p++;
26      cout << *p << endl;
```

Three different notations for the same thing

This notation is specialized to vectors and arrays.

These two work for any random-access iterator.

ITERATOR CLASSIFICATIONS

- Forward iterators:
  - ++ works on iterator
- Bidirectional iterators:
  - Both ++ and – work on iterator
- Random-access iterators:
  - ++, --, and random access all work with iterator
- These are "kinds" of iterators, **not** types!

CONSTANT AND MUTABLE ITERATORS

- Dereferencing operator's behavior dictates
- Constant iterator:
  - * produces read-only version of element
  - Can use *p to assign to variable or output, but cannot change element in container
    - E.g., *p = <anything>; is illegal
- Mutable iterator:
  - *p can be assigned value
  - Changes corresponding element in container
  - i.e.: *p returns an lvalue

REVERSE ITERATORS

- To cycle elements in reverse order
  - Requires container with bidirectional iterators
- Might consider:

```
iterator p;
for (p=container.end();p!=container.begin(); p--)
        cout << *p << " " ;
```

  - But recall: end() is just "sentinel", begin() not!
  - Might work on some systems, but not most

REVERSE ITERATORS CORRECT

- To correctly cycle elements **in reverse order**:

```
reverse_iterator rp;
for (rp=container.rbegin();rp!=container.rend(); rp++)
        cout << *rp << " " ;
```

- rbegin()
  - Returns iterator at last element
- rend()
  - Returns sentinel "end" marker

ITERATOR LIBRARY

- http://en.cppreference.com/w/cpp/iterator

Iterator category					Defined operations
ContiguousIterator	RandomAccessIterator	BidirectionalIterator	ForwardIterator	InputIterator	• read • increment (without multiple passes)
					• increment (with multiple passes)
					• decrement
					• random access
					• contiguous storage
Iterators that fall into one of the above categories and also meet the requirements of OutputIterator are called mutable iterators.					
OutputIterator					• write • increment (without multiple passes)

Iterator primitives

`iterator_traits`	provides uniform interface to the properties of an iterator (class template)

`input_iterator_tag`
`output_iterator_tag`
`forward_iterator_tag`
`bidirectional_iterator_tag`
`random_access_iterator_tag`
	empty class types used to indicate iterator categories (class)

`iterator`	the basic iterator (class template)

Iterator adaptors

`reverse_iterator`	iterator adaptor for reverse-order traversal (class template)
`make_reverse_iterator` (C++14)	creates a `std::reverse_iterator` of type inferred from the argument (function template)
`move_iterator` (C++11)	iterator adaptor which dereferences to an rvalue reference (class template)
`make_move_iterator` (C++11)	creates a `std::move_iterator` of type inferred from the argument (function template)
`back_insert_iterator`	iterator adaptor for insertion at the end of a container (class template)
`back_inserter`	creates a `std::back_insert_iterator` of type inferred from the argument (function template)
`front_insert_iterator`	iterator adaptor for insertion at the front of a container (class template)
`front_inserter`	creates a `std::front_insert_iterator` of type inferred from the argument (function template)
`insert_iterator`	iterator adaptor for insertion into a container (class template)
`inserter`	creates a `std::insert_iterator` of type inferred from the argument (function template)

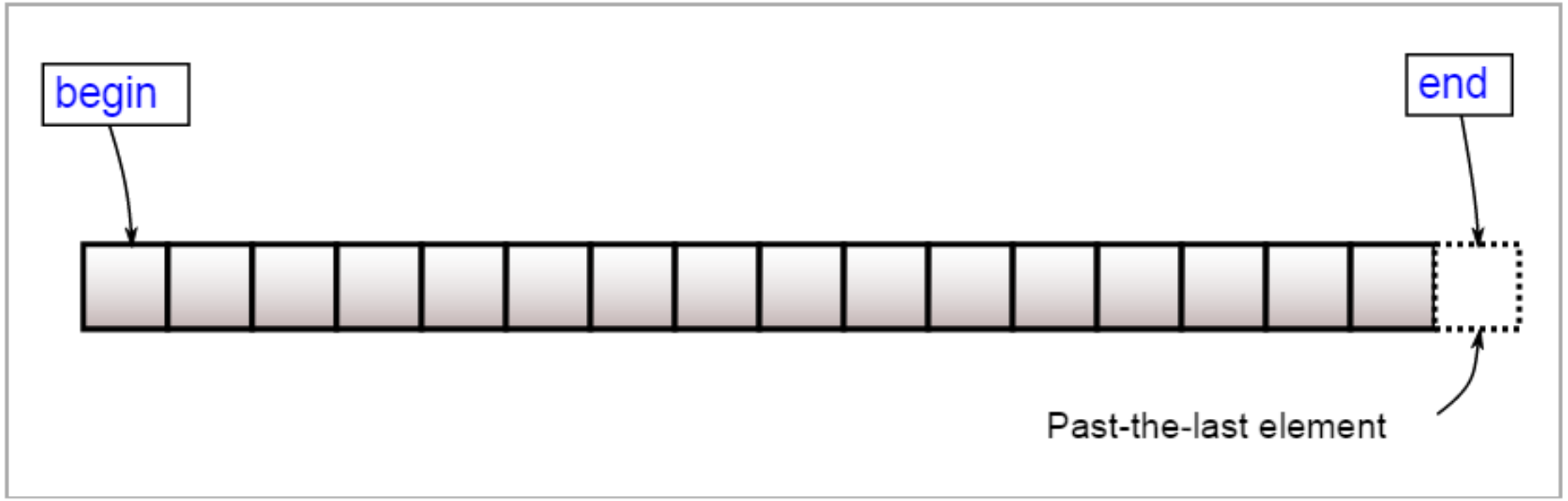
ITERATOR

std::vector::begin, std::vector::cbegin

```
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;      (since C++11)
```

Returns an iterator to the first element of the container.

If the container is empty, the returned iterator will be equal to `end()`.



```
1)  int main ()
2)  {
3)      std::vector<int> vec(3,100);
4)      print_vec(vec);
5)
6)      auto it = vec.begin();
7)      it = vec.insert(it, 200);
8)      print_vec(vec);
9)
10)     vec.insert(it,2,300);
11)     print_vec(vec);
12)
13)     // "it" no longer valid, get a new one:
14)     it = vec.begin();
15)
16)     std::vector<int> vec2(2,400);
17)     vec.insert(it+2, vec2.begin(), vec2.end());
18)     print_vec(vec);
19)
20)     int arr[] = { 501,502,503 };
21)     vec.insert(vec.begin(), arr, arr+3);
22)     print_vec(vec);
23) }
```

```
1) void print_vec(const std::vector<int>& vec)
2) {
3)     for (auto x: vec) {        std::cout << ' ' << x;    }
4)     std::cout << '\n';
5) }
```

```
100 100 100
200 100 100 100
300 300 200 100 100 100
300 300 400 400 200 100 100 100
501 502 503 300 300 400 400 200 100 100 100
```

```
1)  template<class BidirIt>
2)  void my_reverse(BidirIt first, BidirIt last)
3)  {
4)      typename std::iterator_traits<BidirIt>::
            difference_type n = std::distance(first, last);
5)      --n;
6)      while(n > 0)
7)      {
8)          typename std::iterator_traits<BidirIt>::
                        value_type tmp = *first;
9)          *first++ = *--last;
10)         *last = tmp;
11)         n -= 2;
12)     }
13)}
```

```
1)  int main()
2)  {
3)      std::vector<int> v{1, 2, 3, 4, 5};
4)      my_reverse(v.begin(), v.end());
5)      for (int n : v) {
6)          std::cout << n << ' ';
7)      }
8)      std::cout << '\n';
9)
10)     std::list<int> l{1, 2, 3, 4, 5};
11)     my_reverse(l.begin(), l.end());
12)     for (auto n : l) {
13)         std::cout << n << ' ';
14)     }
15)     std::cout << '\n';
16)
17) //   std::istreambuf_iterator<char> i1(std::cin), i2;
18) //   my_reverse(i1, i2); // compilation error
19)
20) }
```

```
5 4 3 2 1
5 4 3 2 1
```

http://en.cppreference.com/w/cpp/iterator/iterator_traits

http://en.cppreference.com/w/cpp/language/dependent_name

COMPILER PROBLEMS

- Some compilers problematic with iterator declarations
- Consider our usage:

```
using std::vector<char>::iterator;
…
iterator p;
```

- Alternatively: `std::vector<char>::iterator p;`
- And others…
  - Try various forms if compiler problematic

CONTAINERS

- Container classes in STL
  - Different kinds of data structures
  - Like lists, queues, stacks
- Each is template class with parameter for particular data type to be stored
  - e.g., Lists of ints, doubles or myClass types
- Each has own iterators
  - One might have bidirectional, another might just have forward iterators
- But all operators and members have same meaning

CORE OF THE C++ STANDARD TEMPLATE LIBRARY

Component	Description
Containers	Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.
Algorithms	Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
Iterators	Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

STANDARD CONTAINERS

- Container: a holder object that stores a collection of other objects (its elements)
  - Implemented as class templates, which allows a great flexibility in the types supported as elements.
  - Manages the storage space for its elements and provides member functions to access them, either directly or through iterators
  - Containers replicate structures very commonly used in programming
    - dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map)…

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

array (since C++11)	static contiguous array (class template)
vector	dynamic contiguous array (class template)
deque	double-ended queue (class template)
forward_list (since C++11)	singly-linked list (class template)
list	doubly-linked list (class template)

Associative containers

Associative containers implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

set	collection of unique keys, sorted by keys (class template)
map	collection of key-value pairs, sorted by keys, keys are unique (class template)
multiset	collection of keys, sorted by keys (class template)
multimap	collection of key-value pairs, sorted by keys (class template)

Unordered associative containers

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ($O(1)$ amortized, $O(n)$ worst-case complexity).

unordered_set (since C++11)	collection of unique keys, hashed by keys (class template)
unordered_map (since C++11)	collection of key-value pairs, hashed by keys, keys are unique (class template)
unordered_multiset (since C++11)	collection of keys, hashed by keys (class template)
unordered_multimap (since C++11)	collection of key-value pairs, hashed by keys (class template)

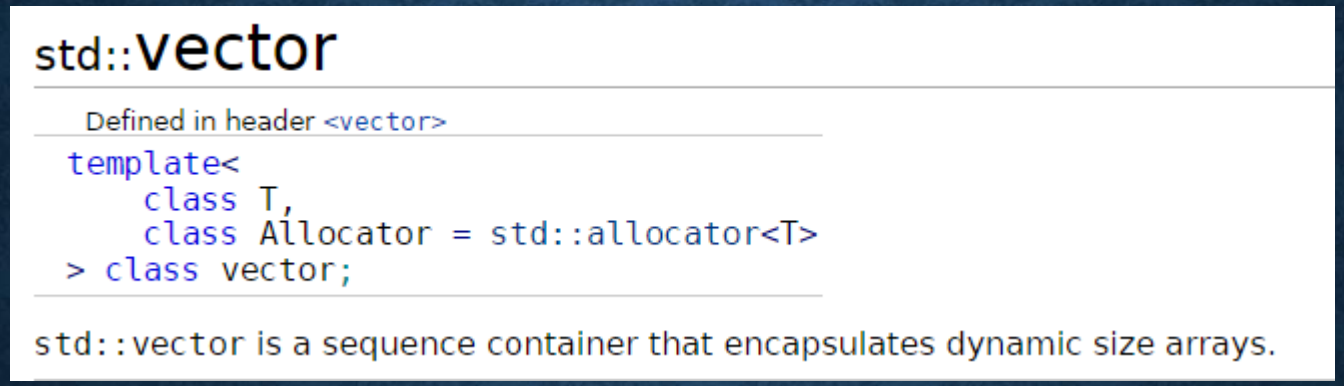
Container adaptors

Container adaptors provide a different interface for sequential containers.

stack	adapts a container to provide stack (LIFO data structure) (class template)
queue	adapts a container to provide queue (FIFO data structure) (class template)
priority_queue	adapts a container to provide priority queue (class template)

SEQUENCE CONTAINER: VECTOR

- http://en.cppreference.com/w/cpp/container/vector



std::vector

Defined in header <vector>

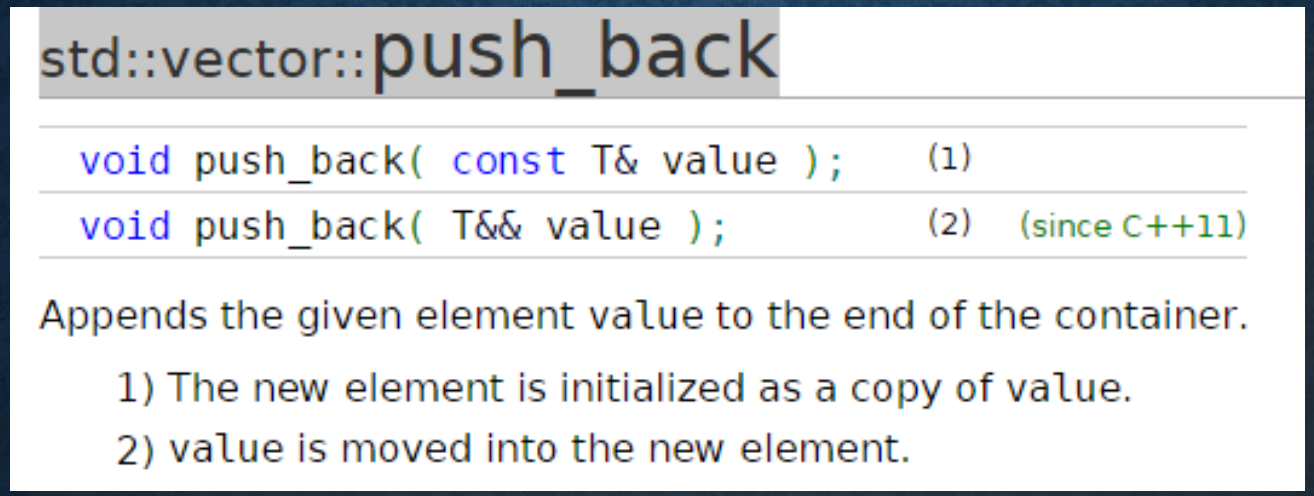
```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

std::vector is a sequence container that encapsulates dynamic size arrays.

- Check to see member function: push_back()

STD::VECTOR::PUSH_BACK

- http://en.cppreference.com/w/cpp/container/vector/push_back



std::vector::push_back

```
void push_back( const T& value );     (1)
void push_back( T&& value );          (2)   (since C++11)
```

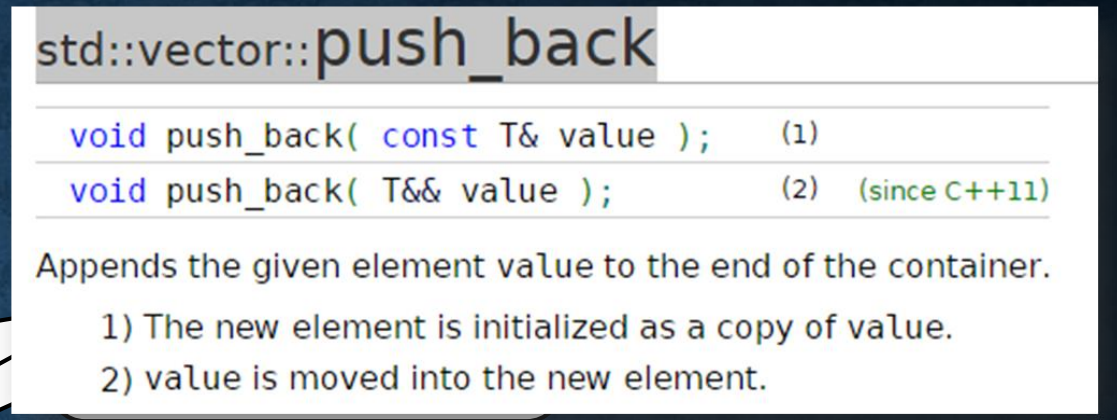
Appends the given element value to the end of the container.

1) The new element is initialized as a copy of value.

2) value is moved into the new element.

Rvalue reference:
http://en.cppreference.com/w/cpp/language/reference

```
1)  int main()
2)  {
3)      std::vector<std::string> numbers;
4)
5)      numbers.push_back("abc");
6)      std::string s = "def";
7)      numbers.push_back(std::move(s));
8)
9)      std::cout << "vector holds: ";
10)     for (auto&& i : numbers) std::cout << std::quoted(i) << ' ';
11)     std::cout << "\nMoved-from string holds " << std::quoted(s) << '\n';
12)}
```



std::vector::push_back

```
void push_back( const T& value );     (1)
void push_back( T&& value );          (2)   (since C++11)
```

Appends the given element value to the end of the container.

1) The new element is initialized as a copy of value.
2) value is moved into the new element.

en.cppreference.com/w/cpp/language/reference

```
1)  int main()
2)  {
3)      std::vec
4)
5)      numbe
6)      std::strin
7)      numbers.push_back(std::move(s));
8)
9)      std::cout << "vector holds: ";
10)     for (auto&& i : numbers) std::cout << std::quoted(i) << ' ';
11)     std::cout << "\nMoved-from string holds " << std::quoted(s) << '\n';
12)}
```

std::move

Defined in header <utility>

```
template< class T >
typename std::remove_reference<T>::type&& move( T&& t );          (since C++11)
                                                                  (until C++14)
template< class T >
constexpr typename std::remove_reference<T>::type&& move( T&& t ); (since C++14)
```

`std::move` is used to *indicate* that an object `t` may be "moved from", i.e. allowing the efficient transfer of resources from `t` to another object.

In particular, `std::move` produces an xvalue expression that identifies its argument `t`. It is exactly equivalent to a static_cast to an rvalue reference type.

en.cppreference.com/w/cpp/language/reference

```
1)  int main()
2)  {
3)      std::string str = "Hello";
4)      std::vector<std::string> v;
5)
6)      // uses the push_back(const T&) overload, which means
7)      // we'll incur the cost of copying str
8)      v.push_back(str);
9)      std::cout << "After copy, str is \"" << str << "\"\n";
10)
11)     // uses the rvalue reference push_back(T&&) overload,
12)     // which means no strings will be copied; instead, the contents
13)     // of str will be moved into the vector.  This is less
14)     // expensive, but also means str might now be empty.
15)     v.push_back(std::move(str));
16)     std::cout << "After move, str is \"" << str << "\"\n";
17)
18)     std::cout << "The contents of the vector are \"" << v[0]  << "\", \"" << v[1] << "\"\n";
19) }
```

```
After copy, str is "Hello"
After move, str is ""
The contents of the vector are "Hello", "Hello"
```

```
1)  int main()
2)  {
3)      std::vector<std::string> numbers;
4)
5)      numbers.push_back("abc");
6)      std::string s = "def";
7)      numbers.push_back(std::move(s));
8)
9)      std::cout << "vector holds: ";
10)     for (auto&& i : numbers) std::cout << std::quoted(i) << ' ';
11)     std::cout << "\nMoved-from string holds " << std::quoted(s) << '\n';
12)}
```

AUTO SPECIFIER

- Specifies that the type of the variable that is being declared will be automatically deduced from its initializer.
- For functions, specifies that the return type is a trailing return type or will be deduced from its return statements.
- Example

```
1) auto a = 1 + 2;
2) std::cout << "type of a: " << typeid(a).name() << '\n';
3) auto c = {1, 2};
4) std::cout << "type of c: " << typeid(c).name() << '\n';
```

```
type of a: int
type of c: std::initializer_list<int>
```

TYPEID OPERATOR

- Queries information of a type
- The typeid expression is lvalue expression which refers to an object with static storage duration, of the polymorphic type const std::type_info or of some type derived from it.
- std::type_info
  - …
  - The class type_info holds implementation-specific information about a type, including the name of the type and means to compare two types for equality or collating order.

RANGE-BASED FOR LOOP

- Executes a for loop over a range
- Syntax： attr(optional) for (range_declaration : range_expression) loop_statement
- Examples:

```
1)   std::vector<int> v = {0, 1, 2, 3, 4, 5};
2)    for(const int &i : v) // access by const reference
3)          std::cout << i << ' ';
4)   for(auto i: v) // access by value, the type of i is int
5)     std::cout << i << ' ';
6)   for(auto&& i: v) // access by reference, the type of i is int&
7)          std::cout << i << ' ';
8)   for(int n: {0, 1, 2, 3, 4, 5}) // the initializer may be a braced-init-list
9)          std::cout << n << ' ';
10)  int a[] = {0, 1, 2, 3, 4, 5};
11)    for(int n: a)  // the initializer may be an array
12)        std::cout << n << ' ';
```

```
1)  int main()
2)  {
3)      std::vector<std::string> numbers;
4)
5)      numbers.push_back("abc");
6)      std::string s = "def";
7)      numbers.push_back(std::move(s));
8)
9)      std::cout << "vector holds: ";
10)     for (auto&& i : numbers) std::cout << std::quoted(i) << ' ';
11)     std::cout << "\nMoved-from string holds " << std::quoted(s) << '\n';
12)}
```

STD::QUOTED

Standard library header <iomanip>

This header is part of the Input/output manipulators library.

Definitions

resetiosflags	clears the specified ios_base flags (function)
setiosflags	sets the specified ios_base flags (function)
setbase	changes the base used for integer I/O (function)
setfill	changes the fill character (function template)
setprecision	changes floating-point precision (function)
setw	changes the width of the next input/output field (function)
get_money (C++11)	parses a monetary value (function template)
put_money (C++11)	formats and outputs a monetary value (function template)
get_time (C++11)	parses a date/time value of specified format (function template)
put_time (C++11)	formats and outputs a date/time value according to the specified format (function template)
quoted (C++14)	inserts and extracts quoted strings with embedded spaces (function template)

STD::QUOTED

```
in:  'String with spaces, and embedded "quotes" too'
stored as '"String with spaces, and embedded \"quotes\" too"'
out:  'String with spaces, and embedded "quotes" too'
```

- Allows insertion and extraction of quoted strings

```
template< class CharT >
/*unspecified*/ quoted(const CharT* s,
                        CharT delim=CharT('"'), CharT escape=CharT('\\'));
```

- Example

```
1)  std::stringstream ss;
2)  std::string in = "String with spaces, and embedded \"quotes\" too";
3)  std::string out;
4)  ss << std::quoted(in);
5)  std::cout << "in:  '" << in << "'\n"   << "stored as '" << ss.str() << "'\n";
6)  ss >> std::quoted(out);
7)  std::cout << "out: '" << out << "'\n";
```

```
1)  int main()
2)  {
3)      std::vector<std::string> numbers;
4)
5)      numbers.push_back("abc");
6)      std::string s = "def";
7)      numbers.push_back(std::move(s));
8)
9)      std::cout << "vector holds: ";
10)     for (auto&& i : numbers) std::cout << std::quoted(i) << ' ';
11)     std::cout << "\nMoved-from string holds " << std::quoted(s) << '\n';
12)}
```

Output:

```
vector holds: "abc" "def"
Moved-from string holds ""
```

en.cppreference.com/w/cpp/language/reference

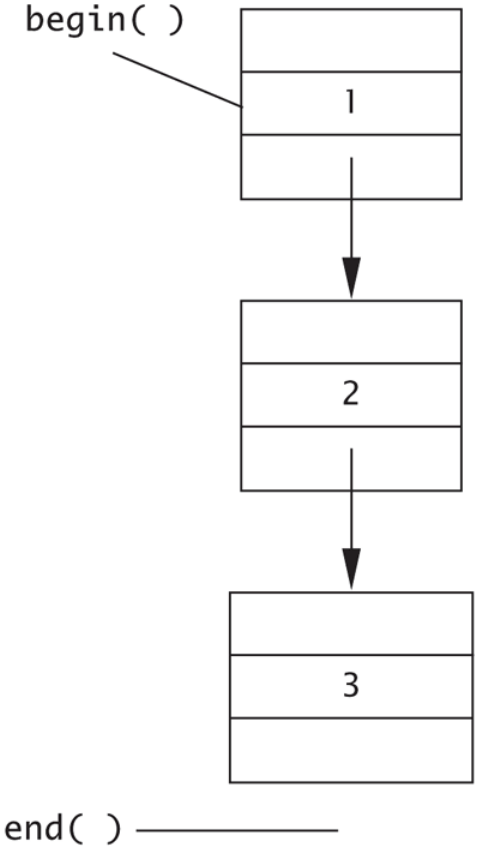
SEQUENTIAL CONTAINERS

- Arranges list data
  - 1st element, next element, … to last element
- Linked list is sequential container
  - Earlier linked lists were "singly linked lists"
    - One link per node
- STL has no "singly linked list"
  - Only "doubly linked list": template class *list*

DISPLAY 19.4 TWO KINDS OF LISTS

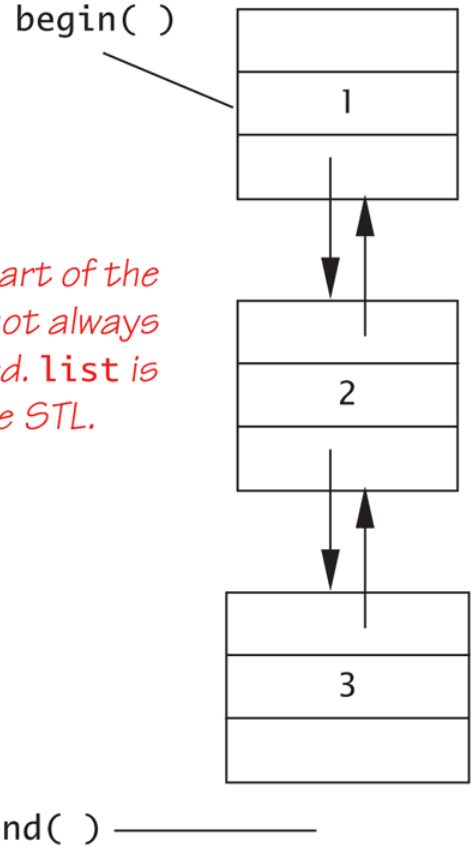
Display 19.4 Two Kinds of Lists

`slist`: *A singly linked list*
++ defined; -- not defined



`slist` *is not part of the STL and may not always be implemented.* `list` *is part of the STL.*

`list`: *A doubly linked list*
Both ++ and -- defined



DISPLAY 19.5 USING THE LIST TEMPLATE CLASS

```
7   int main( )
8   {
9       list<int> listObject;

10      for (int i = 1; i <= 3; i++) listObject.push_back(i);

12      cout << "List contains:\n";
13      list<int>::iterator iter;
14      for (iter = listObject.begin(); iter != listObject.end(); iter++)
15          cout << *iter << " ";
16      cout << endl;
17      cout << "Setting all entries to 0:\n";
18      for (iter = listObject.begin(); iter != listObject.end(); iter++)   *iter= 0;

19      cout << "List now contains:\n";
20      for (iter = listObject.begin(); iter != listObject.end(); iter++)
21          cout << *iter << " ";
22      cout << endl;

23      return 0;
24  }
```

SAMPLE DIALOGUE
List contains:
1 2 3
Setting all entries to 0:
List now contains:
0 0 0

ASSOCIATIVE CONTAINER: SET

std::set

Defined in header <set>

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

std::set is an associative container that contains a sorted set of unique objects of type Key. Sorting is done using the key comparison function Compare. Search, removal, and insertion operations have logarithmic complexity. Sets are usually implemented as red-black trees.

- Check to see set::find()
- Stores elements without repetition
- Insertion places element in set
  - Each element is own key
- Capabilities:
  - Add elements
  - Delete elements
  - Ask if element is in set

```
1)  #include <iostream>
2)  #include <set>
3)
4)  int main()
5)  {
6)      std::set<int> example = {1, 2, 3, 4};
7)
8)      auto search = example.find(2);
9)      if(search != example.end()) {
10)         std::cout << "Found " << (*search) << '\n';
11)     }
12)     else {
13)         std::cout << "Not found\n";
14)     }
15)}
```

SET::FIND()

- http://en.cppreference.com/w/cpp/container/set/find

std::set::find

```
iterator find( const Key& key );                                  (1)
const_iterator find( const Key& key ) const;                      (2)
template< class K > iterator find( const K& x );                  (3)  (since C++14)
template< class K > const_iterator find( const K& x ) const;      (4)  (since C++14)
```

1,2) Finds an element with key equivalent to key.

3,4) Finds an element with key that compares *equivalent* to the value x. This overload only participates in overload resolution if the qualified-id `Compare::is_transparent` is valid and denotes a type. It allows calling this function without constructing an instance of Key

PROGRAM USING THE SET TEMPLATE CLASS

```
7   int main( )
8   {
9       set<char> s;
10      s.insert('A');
11      s.insert('D');
12      s.insert('D');
13      s.insert('C');
14      s.insert('C');
15      s.insert('B');

16      cout << "The set contains:\n";
17      set<char>::const_iterator p;
18      for (p = s.begin( ); p != s.end( ); p++)  cout << *p << " ";
20            cout << endl;
21     cout << "Set contains 'C': ";
22     if (s.find('C')==s.end( )) cout << " no " << endl;
24     else cout << " yes " << endl;
```

```
27      cout << "Removing C.\n";
28      s.erase('C');
29      for (p = s.begin( ); p != s.end( ); p++)
30          cout << *p << " ";
31      cout << endl;

32        cout << "Set contains 'C': ";
33        if (s.find('C')==s.end( ))
34            cout << " no " << endl;
35        else  cout << " yes " << endl;
37      return 0;
38  }
```

```
The set contains:
A B C D
Set contains 'C': yes
Removing C.
A B D
Set contains 'C': no
```

MORE SET TEMPLATE CLASS

- Designed to be efficient
  - Stores values in sorted order
  - Can specify order:
    set<T, *Ordering*> s;
    - Ordering is well-behaved ordering relation that returns bool
    - None specified: use "<" relational operator

```
1)  // constructing sets
2)  #include <iostream>
3)  #include <set>

4)  int main ()
5)  {
6)    std::set<int> first;  // empty set of ints

7)    int myints[]= {10,20,30,40,50};
8)    std::set<int> second (myints,myints+5);       // range
9)    std::set<int> third (second);                 // a copy of second
10)   std::set<int> fourth (second.begin(), second.end());  // iterator constructor.
11)   std::set<int, classcomp> fifth;                 // class as Compare

12)   bool(*fn_pt)(int,int) = fncomp;
13)   std::set<int,bool(*)(int,int)> sixth (fn_pt);  // function pointer as Compare
14)   return 0;
15) }
```

```
bool fncomp (int lhs, int rhs) {return lhs<rhs;}

struct classcomp {
  bool operator() (const int& lhs, const int& rhs) const
  {return lhs<rhs;}
};
```

STD::MULTISET

```
template < class T,                    // multiset::key_type/value_type
           class Compare = less<T>,    // multiset::key_compare/value_compare
           class Alloc = allocator<T> >  // multiset::allocator_type
           > class multiset;
```

- Store elements following a specific order, and where multiple elements can have equivalent values.

```
1)  // multiset::find
2)  int main ()
3)  {
4)    std::multiset<int> mymultiset;
5)    std::multiset<int>::iterator it;

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

8)    it=mymultiset.find(20);
9)    mymultiset.erase (it);
10)   mymultiset.erase (mymultiset.find(40));

11)   std::cout << "mymultiset contains:";
12)   for (it=mymultiset.begin(); it!=mymultiset.end(); ++it)    std::cout << ' ' << *it;
13)   std::cout << '\n';

14)   return 0;
15) }
```

MAP TEMPLATE CLASS

```
template < class Key,                          // map::key_type
           class T,                            // map::mapped_type
           class Compare = less<Key>,                     // map::key_compare
           class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
           > class map;
```

- Store elements formed by a combination of a key value and a mapped value, following a specific order.
- Example map declaration: map<string, int> numberMap;
- Can use [ ] notation to access the map
  - For both storage and retrieval
- Stores in sorted order, like set
  - Second value can have no ordering impact

```
1)  // accessing mapped values
2)  #include <iostream>
3)  #include <map>
4)  #include <string>
5)  int main ()
6)  {
7)    std::map<char, std::string> mymap;
8)    mymap['a']="an element";
9)    mymap['b']="another element";
10)   mymap['c']=mymap['b'];

11)   std::cout << "mymap['a'] is " << mymap['a'] << '\n';
12)   std::cout << "mymap['b'] is " << mymap['b'] << '\n';
13)   std::cout << "mymap['c'] is " << mymap['c'] << '\n';
14)   std::cout << "mymap['d'] is " << mymap['d'] << '\n';

15)   std::cout << "mymap now contains " << mymap.size() << " elements.\n";

16)   return 0;
17) }
```

```
mymap['a'] is an element
mymap['b'] is another element
mymap['c'] is another element
mymap['d'] is
mymap now contains 4 elements.
```

PROGRAM USING THE MAP TEMPLATE CLASS

```
11  map<string, string> planets;
12  planets["Mercury"] = "Hot planet";
13  planets["Venus"] = "Atmosphere of sulfuric acid";
14  planets["Earth"] = "Home";
15  planets["Mars"] = "The Red Planet";
16  planets["Jupiter"] = "Largest planet in our solar syste
17  planets["Saturn"] = "Has rings";
18  planets["Uranus"] = "Tilts on its side";
19  planets["Neptune"] = "1500 mile per hour winds";
20  planets["Pluto"] = "Dwarf planet";
21  cout << "Entry for Mercury - " << planets["Mercury"]
22               << endl << endl;

23  if (planets.find("Mercury") != planets.end())
24       cout << "Mercury is in the map." << endl;
25  if (planets.find("Ceres") == planets.end())
26        cout << "Ceres is not in the map." << endl << end
27  cout << "Iterating through all planets: " << endl;
28  map<string, string>::const_iterator iter;
29  for (iter = planets.begin(); iter != planets.end(); iter++)
30  {
31       cout << iter->first << " - " << iter->second << endl;
32  }
```

SAMPLE DIALOGUE

```
Entry for Mercury - Hot planet

Mercury is in the map.
Ceres is not in the map.

Iterating through all planets:
Earth - Home
Jupiter - Largest planet in our solar system
Mars - The Red Planet
Mercury - Hot planet
Neptune - 1500 mile per hour winds
Pluto - Dwarf planet
Saturn - Has rings
Uranus - Tilts on its side
Venus - Atmosphere of sulfuric acid
```

UNORDERED ASSOCIATIVE CONTAINER: STD::UNORDERED_SET

```
template < class Key,        // unordered_set::key_type/value_type
        class Hash = hash<Key>,        // unordered_set::hasher
        class Pred = equal_to<Key>,     // unordered_set::key_equal
        class Alloc = allocator<Key>     // unordered_set::allocator_type
        > class unordered_set;
```

- Store unique elements in no particular order, and which allow for fast retrieval of individual elements based on their value
  - Keys are immutable, therefore, the elements in an unordered_set **cannot be modified** once in the container
  - Keys can be inserted and removed.

```
1)  // unordered_set::find

2)  int main ()
3)  {
4)    std::unordered_set<std::string> myset = { "red","green","blue" };

5)    std::string input;
6)    std::cout << "color? ";
7)    getline (std::cin, input);

8)    std::unordered_set<std::string>::const_iterator got = myset.find (input);

9)    if ( got == myset.end() )    std::cout << "not found in myset";
10)   else    std::cout << *got << " is in myset";

11)   std::cout << std::endl;

12)   return 0;
13) }
```

```
1)  // unordered_set::rehash

2)  int main ()
3)  {
4)    std::unordered_set<std::string> myset;

5)    myset.rehash(12);

6)    myset.insert("office");
7)    myset.insert("house");
8)    myset.insert("gym");
9)    myset.insert("parking");
10)   myset.insert("highway");

11)   std::cout << "current bucket_count: " << myset.bucket_count() << std::endl;

12)   return 0;
13) }
```

STD::QUEUE

template <class T, class Container = deque<T> > class queue;

- Queues are a type of container adaptor
  - Designed to operate in a FIFO context (first-in first-out)
    - Elements are
      inserted into one end of the container and
      extracted from the other

```
1)   // queue::push/pop
2)    std::queue<int> myqueue;
3)     int myint;

4)     std::cout << "Please enter some integers (enter 0 to end):\n";

5)     do {
6)       std::cin >> myint;
7)       myqueue.push (myint);
8)     } while (myint);

9)     std::cout << "myqueue contains: ";
10)   while (!myqueue.empty())
11)   {
12)     std::cout << ' ' << myqueue.front();
13)     myqueue.pop();
14)   }
15)   std::cout << '\n';
```

CONTAINER ADAPTOR: STD::PRIORITY_QUEUE

- http://en.cppreference.com/w/cpp/container/priority_queue

std::priority_queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

A priority queue is a container adaptor that provides constant time extraction of the largest (by default) element, at the expense of logarithmic insertion.

```
1)  template<typename T> void print_queue(T& q) {
2)      while(!q.empty()) {
3)          std::cout << q.top() << " ";
4)          q.pop();
5)      }
6)      std::cout << '\n';
7)  }
8)
9)  int main() {
10)     std::priority_queue<int> q;
11)
12)     for(int n :{1,8,5,6,3,4,0,9,3,2})        q.push(n);
13)      print_queue(q);
14)
15)     std::priority_queue<int, std::vector<int>, std::greater<int> > q2;
16)      for(int n :{1,8,5,6,3,4,0,9,3,2})        q2.push(n);
17)
18)     print_queue(q2);
19) }
```

Output:

```
9 8 6 5 4 3 3 2 1 0
0 1 2 3 3 4 5 6 8 9
```

GENERIC ALGORITHMS

- Basic template functions
- Recall algorithm definition:
  - Set of instructions for performing a task
  - Can be represented in any language
  - Typically thought of in "pseudocode"
  - Considered "abstraction" of code
    - Gives important details, but not find code details
- STL's algorithms in template functions:
  - Certain details provided only
    - Therefore considered "generic algorithms"

ALGORITHMS LIBRARY

- http://www.cplusplus.com/reference/algorithm/
- http://en.cppreference.com/w/cpp/algorithm
- A collection of functions especially designed to be used on ranges of elements
  - A range is any sequence of objects that can be accessed through iterators or pointers
    - Such as an array or an instance of some of the STL containers

STD::GENERATE

```
template <class ForwardIterator, class Generator>
    void generate (ForwardIterator first, ForwardIterator last, Generator gen);
```

- Generate values for range with function
  - Assigns the value returned by successive calls to gen to the elements in the range [first,last).

```
1)  int RandomNumber () { return (std::rand()%100); } // function generator:
2)  struct c_unique {
3)    int current;
4)    c_unique() {current=0;}
5)    int operator() {return ++current;}
6)  } UniqueNumber;   // class generator:

7)  int main () {
8)    std::srand ( unsigned ( std::time(0) ) );
9)    std::vector<int> myvector (8);
10)   std::generate (myvector.begin(), myvector.end(), RandomNumber);
11)   std::cout << "myvector contains:";
12)   for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)    std::cout << ' ' << *it;
13)   std::cout << '\n';
14)   std::generate (myvector.begin(), myvector.end(), UniqueNumber);
15)   std::cout << "myvector contains:";
16)   for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)    std::cout << ' ' << *it;
17)   std::cout << '\n';
18)    return 0;
19) }
```

```
// generate algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::generate
#include <vector>       // std::vector
#include <ctime>        // std::time
#include <cstdlib>      // std::rand, std::srand
```

```
myvector contains: 57 87 76 66 85 54 17 15
myvector contains: 1 2 3 4 5 6 7 8
```

STD::MAX

- http://en.cppreference.com/w/cpp/algorithm/max

std::max

Defined in header <algorithm>

```
template< class T >
const T& max( const T& a, const T& b );                                  (1)   (until C++14)
template< class T >
constexpr const T& max( const T& a, const T& b );                              (since C++14)

template< class T, class Compare >
const T& max( const T& a, const T& b, Compare comp );                    (2)   (until C++14)
template< class T, class Compare >
constexpr const T& max( const T& a, const T& b, Compare comp );                (since C++14)

template< class T >
T max( std::initializer_list<T> ilist );                                 (3)   (since C++11)
                                                                               (until C++14)
template< class T >
constexpr T max( std::initializer_list<T> ilist );                             (since C++14)

template< class T, class Compare >
T max( std::initializer_list<T> ilist, Compare comp );                   (4)   (since C++11)
                                                                               (until C++14)
template< class T, class Compare >
constexpr T max( std::initializer_list<T> ilist, Compare comp );               (since C++14)
```

Returns the greater of the given values.

1-2) Returns the greater of a and b.

3-4) Returns the greatest of the values in initializer list `ilist`.

The (1,3) versions use `operator<` to compare the values, the (2,4) versions use the given comparison function comp.

```
larger of 1 and 9999: 9999
larger of 'a', and 'b': b
longest of "foo", "bar", and "hello": hello
```

```
1) #include <algorithm>
2) #include <iostream>
3) #include <string>

4) bool fc(const std::string& s1, const std::string& s2)
5) {
6)     return s1.size() < s2.size();
7) }
8)
9) int main()
10){
11)    std::cout << "larger of 1 and 9999: " << std::max(1, 9999) << '\n'
12)              << "larger of 'a', and 'b': " << std::max('a', 'b') << '\n'
13)              << "longest of \"foo\", \"bar\", and \"hello\": " <<
14)                 std::max( { "foo", "bar", "hello" }, fc) << '\n';
15)}
```

STD::BINARY_SEARCH

- http://en.cppreference.com/w/cpp/algorithm/binary_search

std::binary_search

Defined in header <algorithm>

```
template< class ForwardIt, class T >
bool binary_search( ForwardIt first, ForwardIt last, const T& value );                    (1)

template< class ForwardIt, class T, class Compare >
bool binary_search( ForwardIt first, ForwardIt last, const T& value, Compare comp );     (2)
```

```
1)  #include <iostream>
2)  #include <algorithm>
3)  #include <vector>
4)
5)  int main()
6)  {
7)      std::vector<int> haystack {1, 3, 4, 5, 9};  //already be sorted
8)      std::vector<int> needles {1, 2, 3};
9)
10)     for (auto needle : needles) {
11)         std::cout << "Searching for " << needle << '\n';
12)         if (std::binary_search(haystack.begin(), haystack.end(), needle))
13)         {
14)             std::cout << "Found " << needle << '\n';
15)         }
16)         else {          std::cout << "no dice!\n";       }
17)     }
18) }
```

```
Searching for 1
Found 1
Searching for 2
no dice!
Searching for 3
Found 3
```

STD::SORT

- http://en.cppreference.com/w/cpp/algorithm/sort

std::sort

Defined in header <algorithm>

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );                          (1)

template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );            (2)
```

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The first version uses `operator<` to compare the elements, the second version uses the given comparison function object comp.

- Sorts the elements in the range [first,last)

```
1)   // sort algorithm example
2)   #include <iostream>     // std::cout
3)   #include <algorithm>    // std::sort
4)   #include <vector>      // std::vector

5)   bool myfunction (int i,int j) { return (i<j); }
6)   struct myclass {  bool operator() (int i,int j) { return (i<j);} } myobject;

7)   int main ()
8)   {
9)     int myints[] = {32,71,12,45,26,80,53,33};
10)    std::vector<int> myvector (myints, myints+8);              // 32 71 12 45 26 80 53 33

11)    // using default comparison (operator <):
12)    std::sort (myvector.begin(), myvector.begin()+4);         //(12 32 45 71)26 80 53 33

13)    // using function as comp
14)    std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)

15)    // using object as comp
16)    std::sort (myvector.begin(), myvector.end(), myobject);    //(12 26 32 33 45 53 71 80)

17)    // print out content:
18)    std::cout << "myvector contains:";
19)    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)    std::cout << ' ' << *it;
20)    std::cout << '\n';

21)    return 0;
22)  }
```

myvector contains: 12 26 32 33 45 53 71 80

DISPLAY 19.17 THE GENERIC find FUNCTION (2 OF 3)

```
12      vector<char> line;

13      cout << "Enter a line of text:\n";
14      char next;
15      cin.get(next);
16      while (next != '\n')
17      {
18          line.push_back(next);
19          cin.get(next);
20      }
21      vector<char>::const_iterator where;
22      where = find(line.begin( ), line.end( ), 'e');
23      //where is located at the first occurrence of 'e' in v.

24      vector<char>::const_iterator p;
25      cout << "You entered the following before you entered your
          first e:\n";
26      for (p = line.begin( ); p != where; p++)
27          cout << *p;
28      cout << endl;

29      cout << "You entered the following after that:\n";
30      for (p = where; p != line.end( ); p++)
31          cout << *p;
32      cout << endl;

33      cout << "End of demonstration.\n";
34      return 0;
```

SAMPLE DIALOGUE 1

Enter a line of text
A line of text.
You entered the following before you entered your first e:
A lin
You entered the following after that:
e of text.
End of demonstration.

SAMPLE DIALOGUE 2

Enter a line of text
I will not!
You entered the following before you entered your first e:
I will not!
You entered the following after that:

End of demonstration

FUNCTION OBJECT

- A function object is an object to be invoked or called as if it were an ordinary function
- Why function object?
  - See examples at next page!

```
1) /* qsort() callback function, returns
            < 0 if a < b,
            > 0 if a > b,
            0 if a == b */
2) int compareInts(const void* a, const void* b)
3) {
4)     return ((int) a > (int) b) - ((int) a < (int) b);
5) }
6) ...
7) // void qsort(void *base, size_t nel, size_t width,
            int (*compar)(const void *, const void *));
8) ...
9) int main(void)
10){
11)    int items[] = { 4, 3, 1, 2 };
12)    qsort(items, sizeof(items) / sizeof(items[0]),
            sizeof(items[0]), compareInts);
13)    return 0;
14)}
```

```
1)  // comparator predicate: returns true if a < b, false
    otherwise
2)  struct IntComparator
3)  {
4)    bool operator()(const int &a, const int &b) const
5)    {
6)      return a < b;
7)    }
8)  };
9)  ...
10) // An overload of std::sort is:
11) template <class RandomIt, class Compare>
12) void sort(RandomIt first, RandomIt last, Compare comp);
13) ...
14) int main()
15) {
16)     std::vector<int> items { 4, 3, 1, 2 };
17)     std::sort(items.begin(), items.end(), IntComparator());
18)     return 0;
19) }
```

```
1)  struct CompareBy
2)  {
3)      const std::string SORT_FIELD;
4)      CompareBy(const std::string& sort_field="name")
5)        : SORT_FIELD(sort_field)   {       /* validate sort_field */   }
6)
7)      bool operator()(const Employee& a, const Employee& b)
8)      {
9)          if (SORT_FIELD == "name")
10)             return a.name < b.name;
11)         else if (SORT_FIELD == "age")
12)             return a.age < b.age;
13)         else if (SORT_FIELD == "idnum")
14)             return a.idnum < b.idnum;
15)         else
16)             /* throw exception or something */
17)     }
18) };
```

```
1)  int main()
2)  {
3)      std::vector<Employee> emps;
4)
5)      /* code to populate database */
6)
7)      // Sort the database by employee ID number
8)      std::sort(emps.begin(), emps.end(),
        CompareBy("idnum"));
9)
10)     return 0;
11) }
```

EFFICIENCY

- STL designed with efficiency as important consideration
  - Strives to be optimally efficient
- Example: set, map elements stored in sorted order for fast searches
- Template class member functions:
  - Guaranteed maximum running time
  - Called "Big-O" notation, an "efficiency"-rating

RUNNING TIMES

- How fast is program?
  - "Seconds"?
  - Consider: large input? .. small input?
- Produce "table"
  - Based on input size
  - Table called "function" in math
    - With arguments and return values!
  - Argument is input size: T(10), T(10,000), …
- Function **T** is called "**running time**"

Some Values of a Running Time Function

INPUT SIZE	RUNNING TIME
10 numbers	2 seconds
100 numbers	2.1 seconds
1,000 numbers	10 seconds
10,000 numbers	2.5 minutes

CONSIDER SORTING PROGRAM

- Faster on smaller input set?
  - Perhaps
  - Might depend on "state" of set
    - "Mostly" sorted already?
- Consider worst-case running time
  - T(N) is time taken by "hardest" list
    - List that takes longest to sort

COUNTING OPERATIONS

- T(N) given by formula, such as:

  $T(N) = 5N + 5$

  - "On inputs of size N program runs for 5N + 5 time units"
- Must be "computer-independent"
  - Doesn't matter how "fast" computers are
  - Can't count "time"
  - Instead count "operations"

COUNTING OPERATIONS EXAMPLE

- ```
  int I = 0;
  bool found = false;
  while (( I < N) && !found)
          if (a[I] == target)
                  found = true;
          else
                  I++;
  ```
- 5 operations per loop iteration:
  <, &&, !, [ ], ==, ++
- After N iterations, final three: <, &&, !
- So: 5N+5 operations when target not found
  ```

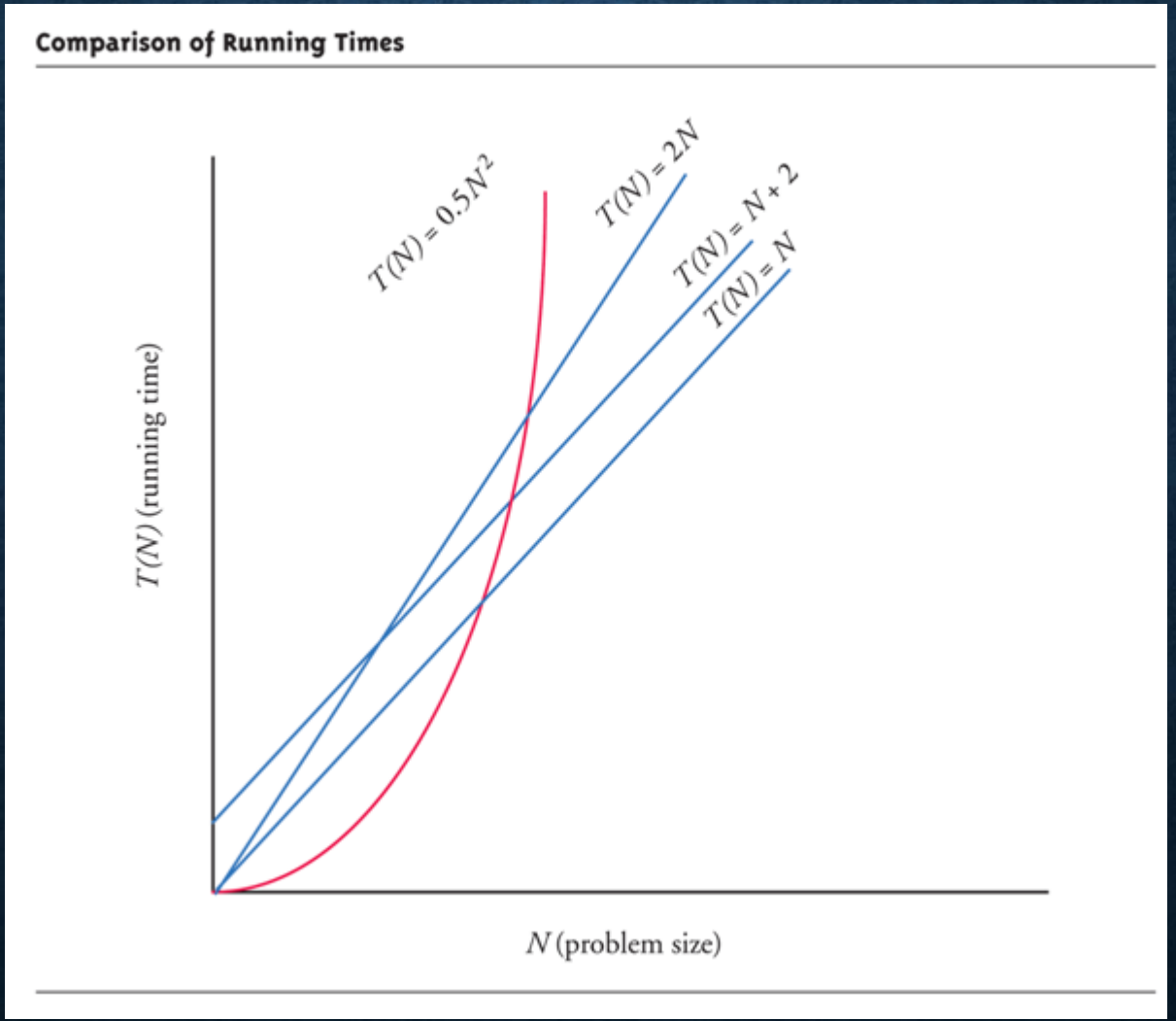
BIG-O NOTATION

- Recall: 5N+5 operations in "worst-case"
- Expressed in "Big-O" notation
  - Some constant "c" factor where
    c(5N+5) is actual running time
    - c different on different systems
  - We say code runs in time O(5N+5)
  - But typically only consider "highest term"
    - Term with highest exponent
  - O(N) here

BIG-O TERMINOLOGY

- Linear running time:
  - O(N)—directly proportional to input size N
- Quadratic running time:
  - $O(N^2)$
- Logarithmic running time:
  - O(log N)
    - Typically "log base 2"
    - Very fast algorithms!

DISPLAY 19.16
COMPARISON OF RUNNING TIMES



CONTAINER ACCESS RUNNING TIMES

- O(1) - constant operation always:
  - Vector inserts to front or back
  - deque inserts
  - list inserts
- O(N)
  - Insert or delete of arbitrary element in vector or deque
    - N is number of elements
- O(log N)
  - set or map finding

SORTING ALGORITHMS

- STL contains two template functions:
  1. sort range of elements
  2. merge two sorted ranges of elements
- Guaranteed running time O(N log N)
  - No sort can be faster
  - Function guarantees fastest possible sort

SUMMARY 1

- Iterator is "generalization" of a pointer
  - Used to move through elements of container
- Container classes with iterators have:
  - Member functions end() and begin() to assist cycling
- Main kinds of iterators:
  - Forward, bi-directional, random-access
- Given constant iterator p, *p is read-only version of element

SUMMARY 2

- Given mutable iterator p → *p can be assigned value
- Bidirectional container has reverse iterators allowing reverse cycling
- Main STL containers: list, vector, deque
  - stack, queue: container adapter classes
- set, map, multiset, multimap containers store in sorted order
- STL implements generic algorithms
  - Provide maximum running time guarantees