Object-Oriented Programming
(in C++)

Template and Standard Template Library

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

What Are Templates?

- Templates are C++'s powerful features that allows generic programming
- Templates (function templates and class templates) promote code reusability because one function or class template can be used to generate many functions or classes

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

Function Templates

- Used to produce overloaded functions that perform identical operations on different types of data
  - Programmer writes a single function-template definition
  - Compiler generates separate object-code functions (**function-template specializations**) based on argument types in calls to the function template at compile time

Function-Template Definitions

- Preceded by a template header
  - Keyword `template`
  - List of template parameters
    - keyword `class` or `typename` (interchangeable)
    - Used to specify types of ***arguments***, ***local variables*** and ***return type*** of the function template (any built-in type or user-defined type)
  - Examples

```
template< typename T >
template< class ElementType >
template< typename BorderType, typename Filltype >
```

An Example of Function Templates

```
3  #include <iostream>
4  using namespace std;
5
6  // function template printArray definition
7  template< typename T >
8  void printArray( const T * const array, int count )
9  {
10    for ( int i = 0; i < count; i++ )
11       cout << array[ i ] << " ";
12
13    cout << endl;
14 } // end function template printArray
15
16 int main()
17 {
18    const int aCount = 5; // size of array a
19    const int bCount = 7; // size of array b
20    const int cCount = 6; // size of array c
21
22    int a[ aCount ] = { 1, 2, 3, 4, 5 };
23    double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
24    char c[ cCount ] = "HELLO"; // 6th position for null
25
26    cout << "Array a contains:" << endl;
27
28    // call integer function-template specialization
29    printArray( a, aCount );
30
31    cout << "Array b contains:" << endl;
32
33    // call double function-template specialization
34    printArray( b, bCount );
35
36    cout << "Array c contains:" << endl;
37
38    // call character function-template specializatio
39    printArray( c, cCount );
40 } // end main
```

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

```
void printArray(const int*const array, int count)
{
  for (int i = 0; i < count; i++)
    cout << array[ i ] << " ";

  cout << endl;
}
```

Creates a function-template specialization of `printArray` where `int` replaces `T`

Creates a function-template specialization of `printArray` where `double` replaces `T`

Creates a function-template specialization of `printArray` where `char` replaces `T`

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

Class Templates

- C++ Class Templates are used where we have multiple copies of code for different data types with the same logic
- Class templates are called parameterized types
  - Require one or more type parameters to specify how to customize a “generic class”
- A class template defines a family of classes
- It serves as a class outline, from which specific classes are generated at compile time

Class-Template Definitions

- Class-template definitions are preceded by a header

```
template< typename T >
```

- Type parameter T can be used as a data type in member functions and data members
- Additional type parameters can be specified using a comma-separated list

```
template< typename T1, typename T2 >
```

An Example of Class Templates (1/2)

Stack.h

```
6  template< typename T >
7  class Stack
8  {
9  public:
10    Stack( int = 10 ); // default constructor (Stack size 10)
11
12    // destructor
13    ~Stack()
14    {
15       delete [] stackPtr; // deallocate internal space for Stack
16    } // end ~Stack destructor
17
18    bool push( const T & ); // push an element onto the Stack
19    bool pop( T & ); // pop an element off the Stack
20
21    // determine whether Stack is empty
22    bool isEmpty() const
23    {
24       return top == -1;
25    } // end function isEmpty
26
27    // determine whether Stack is full
28    bool isFull() const
29    {
30       return top == size - 1;
31    } // end function isFull
32
33 private:
34    int size; // # of elements in the Stack
35    int top; // location of the top element (-1 means empty)
36    T *stackPtr; // pointer to internal representation of the Stack
37 }; // end class template Stack
38
```

An Example of Class Templates (2/2)

Stack.cpp

```
39 // constructor template
40 template< typename T >
41 Stack< T >::Stack( int s )
42    : size( s > 0 ? s : 10 ), // validate size
43      top( -1 ), // Stack initially empty
44      stackPtr( new T[ size ] ) // allocate memory for elements
45 {
46    // empty body
47 } // end Stack constructor template
48
49 // push element onto Stack;
50 // if successful, return true; otherwise, return false
51 template< typename T >
52 bool Stack< T >::push( const T &pushValue )
53 {
54    if ( !isFull() )
55    {
56       stackPtr[ ++top ] = pushValue; // place item on Stack
57       return true; // push successful
58    } // end if
59
60    return false; // push unsuccessful
61 } // end function template push
62
63 // pop element off Stack;
64 // if successful, return true; otherwise, return false
65 template< typename T >
66 bool Stack< T >::pop( T &popValue )
67 {
68    if ( !isEmpty() )
69    {
70       popValue = stackPtr[ top-- ]; // remove item from Stack
71       return true; // pop successful
72    } // end if
73
74    return false; // pop unsuccessful
75 } // end function template pop
```

Complete Template Definition in Header File

- Most C++ compilers require the complete definition of a template to appear in the client source-code file that uses the template
  - Templates are often defined in header files
  - For class templates, the member functions are also defined in the header file

A Client of Stack

```
3  #include <iostream>
4  #include "Stack.h" // Stack class template definition
5  using namespace std;
6
7  int main()
8  {
9     Stack< double > doubleStack( 5 ); // size 5
10    double doubleValue = 1.1;
11
12    cout << "Pushing elements onto doubleStack\n";
13
14    // push 5 doubles onto doubleStack
15    while ( doubleStack.push( doubleValue ) )
16    {
17       cout << doubleValue << ' ';
18       doubleValue += 1.1;
19    } // end while
20
21    cout << "\nStack is full. Cannot push " << doubleValue
22       << "\n\nPopping elements from doubleStack\n";
23
24    // pop elements from doubleStack
25    while ( doubleStack.pop( doubleValue ) )
26       cout << doubleValue << ' ';
27
28    cout << "\nStack is empty. Cannot pop\n";
29
30    Stack< int > intStack; // default size 10
31    int intValue = 1;
32    cout << "\nPushing elements onto intStack\n";
33
34    // push 10 integers onto intStack
35    while ( intStack.push( intValue ) )
36    {
37       cout << intValue++ << ' ';
38    } // end while
39
40    cout << "\nStack is full. Cannot push " << intValue
41       << "\n\nPopping elements from intStack\n";
42
43    // pop elements from intStack
44    while ( intStack.pop( intValue ) )
45       cout << intValue << ' ';
46
47    cout << "\nStack is empty. Cannot pop" << endl;
48 } // end main
```

Creates a class-template specialization of `Stack` where `double` replaces `T`

```
class Stack< double > {
public:
   Stack( int = 10 );
   ~Stack() {
      delete [] stackPtr;
   }
   bool push( const double & );
   bool pop(double & );

   bool isEmpty() const {
      return top == -1;
   }

   bool isFull() const {
      return top == size - 1;
   }
private:
   int size;
   int top;
   double *stackPtr;
};
Stack< double >::Stack( int s )
   : size( s > 0 ? s : 10 ),
     top( -1 ),
     stackPtr( new double[ size ] ) {
}
bool Stack< double >::push( const double &pushValue ) {
   if ( !isFull() ) {
      stackPtr[ ++top ] = pushValue;
      return true;
   }
   return false;
}
bool Stack< double >::pop( double &popValue ) {
   if ( !isEmpty() ) {
      popValue = stackPtr[ top-- ];
      return true;
   }
   return false;
}
```

Another Client

```
4  #include <iostream>
5  #include <string>
6  #include "Stack.h" // Stack class template definition
7  using namespace std;
8
9  // function template to manipulate Stack< T >
10 template< typename T >
11 void testStack(
12    Stack< T > &theStack, // reference to Stack< T >
13    T value, // initial value to push
14    T increment, // increment for subsequent values
15    const string stackName ) // name of the Stack< T > object
16 {
17    cout << "\nPushing elements onto " << stackName << '\n';
18
19    // push element onto Stack
20    while ( theStack.push( value ) )
21    {
22       cout << value << ' ';
23       value += increment;
24    } // end while
25
26    cout << "\nStack is full. Cannot push " << value
27       << "\n\nPopping elements from " << stackName << '\n';
28
29    // pop elements from Stack
30    while ( theStack.pop( value ) )
31       cout << value << ' ';
32
33    cout << "\nStack is empty. Cannot pop" << endl;
34 } // end function template testStack
35
36 int main()
37 {
38    Stack< double > doubleStack( 5 ); // size 5
39    Stack< int > intStack; // default size 10
40
41    testStack( doubleStack, 1.1, 1.1, "doubleStack" );
42    testStack( intStack, 1, 1, "intStack" );
43 } // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Creates a function-template specialization of `testStack` where `double` replaces `T`

Creates a function-template specialization of `testStack` where `int` replaces `T`

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

Default Template Types

- A type parameter can specify a default type
- For example,

```
template< typename T = string >
class Stack {
  ...
};
```

```
#include "Stack.h"

int main() {
  Stack<> myStack;
  ...
}
```

Non-type Template Parameters (1/2)

- In the template header, it's possible to use non-type template parameters
- For example,

Treated as a constant

```
template< typename T, int elements = 10 >
class Stack {
  ...
private:
  T stackHolder[ elements ]; // array to hold Stack contents
};
```

Eliminates the execution-time overhead of using new to create the space dynamically

```
#include "Stack.h"

int main() {
  Stack< double, 100 > myStack;
  ...
}
```

Non-type Template Parameters (1/2)

- A non-*type template parameter* shall have one of the following types
  - integral or enumeration type
  - pointer to object or pointer to function
  - lvalue reference to object or lvalue reference to function
  - pointer to member
  - `std::nullptr_t`

Notes on Template Specialization

```
template< typename T, int elements >
class Stack {
  ...
private:
  T stackHolder[ elements ]; // array to hold Stack contents
};
```

```
#include "Stack.h"
#include "Employee.h"

int main() {
  Stack< Employee, 100 > myStack;
  ...
}
```

If class Employee has no default constructor, the compiler will report an error!

- The specialized class will behavior incorrectly if Employee does not provide an assignment operator that properly copies objects

Explicit Specialization: An Example

```
template <class T>
class Formatter {
   T* m_t;
public:
   Formatter(T* t) : m_t(t) { }
   void print() {
      cout << *m_t << endl;
   }
};

template<>
class Formatter<char*> {
   char** m_t;
public:
   Formatter(char** t) : m_t(t) { }
   void print() {
      cout << "Char value: " << **m_t << endl;
   }
};
```

Customized processing

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

The Standard Template Library (STL)

- The *Standard Template Library* (STL) is an addition to the C++ Standard Library
  - Developed at the Hewlett-Packard Labs in the mid-late 90's
  - STL defines powerful, template-based, reuseable components that implement many common data structures and algorithms to process those data structures
  - STL is extremely complex, being optimised for efficiency. We will only scratch the surface
- STL is based on three components
  - Containers, Iterators, and Algorithms

Containers, Iterators, and Algorithms

- Containers are data structures used to store objects of almost any type
  - Examples include linked lists, vectors, and sets
- Iterators encapsulate the mechanisms used to access container elements
  - A simple pointer can be used as an iterator for a standard array
- Algorithms are functions that perform common manipulations on containers
  - Examples include inserting, deleting, searching, sorting, etc.
  - There are around 70 algorithms implemented in the STL

Generic Programming

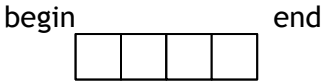
- The STL approach allows general programs to be written so that the code does not depend on the underlying container
  - Such a programming style is called generic programming
- STL generally avoids inheritance and `virtual` functions in favor of using generic programming with templates to achieve better execution-time performance

Containers

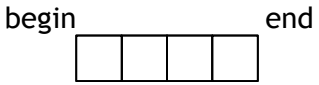
- Containers divided into 3 main categories
  - Sequence containers
    - Essentially linear data structures
  - Associative containers
    - Can be searched quickly using key/value pairs
  - Container adapters
    - Constrained versions of sequence containers
    - Stacks, queues, and priority queues

Sequence Containers

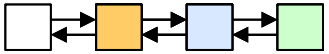
- Examples of sequence STL containers include
  - `vector`



    - Provides random access with constant time insertions and deletions at the end
  - `deque` (double-ended queue)



    - Provides random access with constant time insertions and deletions at the beginning and the end
  - `list` (doubly linked list)



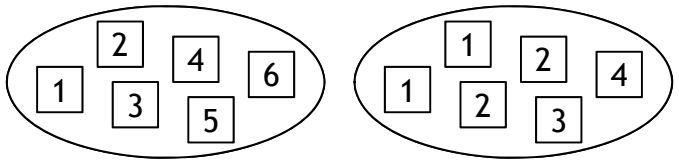
    - Provides linear time access but constant time insertions and deletions at *any* position in the list
  - `string` (acts like a sequence container)
    - Similar to `vector`, but stores only character data

Sequence Containers: Performance Tips

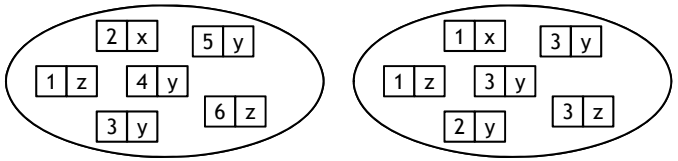
- In general, `deque` has higher overhead than `vector`
- Insertions and deletions in the middle of a `deque` are optimized to minimize the number of elements copied, so it more efficient than a `vector` bus less efficient than a `list` for this kind of modification

Associative Containers

- Examples of associative STL containers include
  - `set`
    - Provides rapid lookup (set membership), no duplicates allowed
  - `multiset`
    - As above but allows duplicates
  - `map`
    - Provides rapid lookup (one-to-one mapping), no duplicates allowed
  - `multimap`
    - As above but allows duplicates



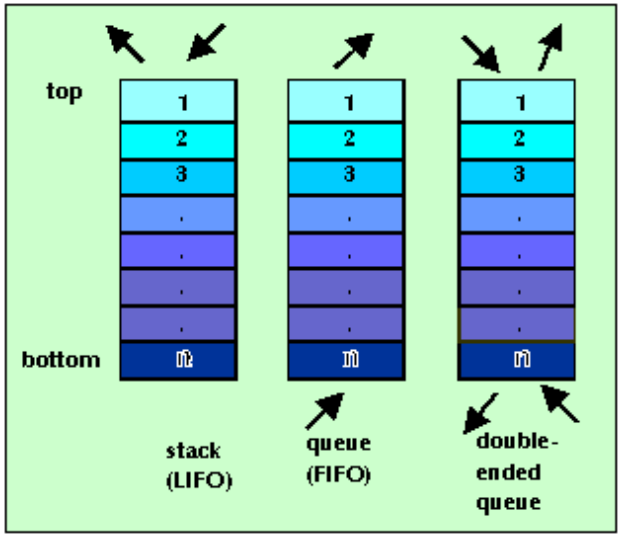
`set` `multiset`



`map` `multimap`

Container Adapters

- Examples of adapter STL containers include
  - `stack`
    - Last-in, first-out (LIFO) data structure
  - `queue`
    - First-in, first-out (FIFO) data structure
  - `priority_queue`
    - Highest-priority element is always the first element out



STL Container Common Functions (1/2)

Member function	Description
default constructor	A constructor to create an empty container. Normally, each container has several constructors that provide different initialization methods for the container.
copy constructor	A constructor that initializes the container to be a copy of an existing container of the same type.
destructor	Destructor function for cleanup after a container is no longer needed.
`empty`	Returns `true` if there are no elements in the container; otherwise, returns `false`.
`insert`	Inserts an item in the container.
`size`	Returns the number of elements currently in the container.
`operator=`	Assigns one container to another.
`operator<`	Returns `true` if the first container is less than the second container; otherwise, returns `false`.
`operator<=`	Returns `true` if the first container is less than or equal to the second container; otherwise, returns `false`.
`operator>`	Returns `true` if the first container is greater than the second container; otherwise, returns `false`.

- Overloaded operators <, <=, >, >=, ==, and != are not provided for `priority_queues`

STL Container Common Functions (2/2)

Member function	Description
`operator>=`	Returns `true` if the first container is greater than or equal to the second container; otherwise, returns `false`.
`operator==`	Returns `true` if the first container is equal to the second container; otherwise, returns `false`.
`operator!=`	Returns `true` if the first container is not equal to the second container; otherwise, returns `false`.
`swap`	Swaps the elements of two containers.
Functions found only in first-class containers	
`max_size`	Returns the maximum number of elements for a container.
`begin`	The two versions of this function return either an `iterator` or a `const_iterator` that refers to the first element of the container.
`end`	The two versions of this function return either an `iterator` or a `const_iterator` that refers to the next position after the end of the container.
`rbegin`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the last element of the container.
`rend`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the next position after the last element of the reversed container.
`erase`	Erases one or more elements from the container.
`clear`	Erases all elements from the container.

STL Container Header Files

	Standard Library container header files	
Sequence containers (First-class containers)	`<vector>`	
	`<list>`	
	`<deque>`	
Container adapters	`<queue>`	Contains both `queue` and `priority_queue`.
	`<stack>`	
Associative containers (First-class containers)	`<map>`	Contains both `map` and `multimap`.
	`<set>`	Contains both `set` and `multiset`.
Near containers	`<valarray>`	
	`<bitset>`	

First-Class Container Common `typedefs`

typedef	Description
`allocator_type`	The type of the object used to allocate the container's memory.
`value_type`	The type of element stored in the container.
`reference`	A reference to the type of element stored in the container.
`const_reference`	A constant reference to the type of element stored in the container. Such a reference can be used only for *reading* elements in the container and for performing `const` operations.
`pointer`	A pointer to the type of element stored in the container.
`const_pointer`	A pointer to a constant of the container's element type.
`iterator`	An iterator that points to an element of the container's element type.
`const_iterator`	A constant iterator that points to the type of element stored in the container and can be used only to *read* elements.
`reverse_iterator`	A reverse iterator that points to the type of element stored in the container. This type of iterator is for iterating through a container in reverse.
`const_reverse_iterator`	A constant reverse iterator that points to the type of element stored in the container and can be used only to *read* elements. This type of iterator is for iterating through a container in reverse.
`difference_type`	The type of the result of subtracting two iterators that refer to the same container (operator `-` is not defined for iterators of `lists` and associative containers).
`size_type`	The type used to count items in a container and index through a sequence container (cannot index through a `list`).

Attention When Using STL Containers

- When preparing to use an STL container, it is important to ensure that the type of element being stored in the container supports a minimum set of functionality
  - When an element is inserted into a container, a copy of that element is made
    - The element type should provide appropriate copy constructor and assignment operator
  - Associative containers and many algorithms (e.g., `sort`) require elements to be compared
    - The element type should provide an equality operator (==) and a less-than operator (<)

Iterators

- Iterators have many features in common with pointers
  - Used to point to the element of first-class containers (and for a few other purposes)
- Dereferencing an iterator returns the element to which it points
- The ++ operation on an iterator moves it to the next element of the container

Iterators: An Example

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> integers1(5);

  // a pointer to a vector<int> element
  vector<int>::iterator it;

  for(it = integers1.begin(); it != integers1.end(); ++it)
    cout << *it << " ";

  return 0;
}
```

- First-class containers provide member functions
  - begin() returns an iterator pointing to the first element
  - end() returns an iterator pointing to the first element past the end of the container (an element that doesn't exist)

Iterators: Another Example

```
3  #include <iostream>
4  #include <iterator> // ostream_iterator and istream_iterator
5  using namespace std;
6
7  int main()
8  {
9     cout << "Enter two integers: ";
10
11    // create istream_iterator for reading int values from cin
12    istream_iterator< int > inputInt( cin );
13
14    int number1 = *inputInt; // read int from standard input
15    ++inputInt; // move iterator to next input value
16    int number2 = *inputInt; // read int from standard input
17
18    // create ostream_iterator for writing int values to cout
19    ostream_iterator< int > outputInt( cout );
20
21    cout << "The sum is: ";
22    *outputInt = number1 + number2; // output result to cout
23    cout << endl;
24 } // end main
```

```
Enter two integers: 12 25
The sum is: 37
```

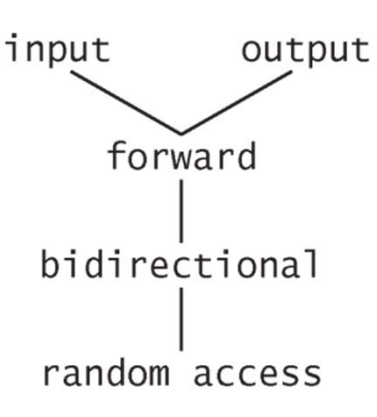
Common Programming Errors

- Attempting to deference an iterator positioned outside its container is a runtime logic error
  - The iterator returned by `end` cannot be dereferenced or incremented
- Attempting to create a non-`const` iterator for a `const` container is a compilation error
  - Because the container should not be modified
- Attempting to use non-`const` member functions on a container element through a reference that is returned by dereferencing a `const` iterator is a compilation error
  - Because the reference is a `const` reference

Iterator Categories

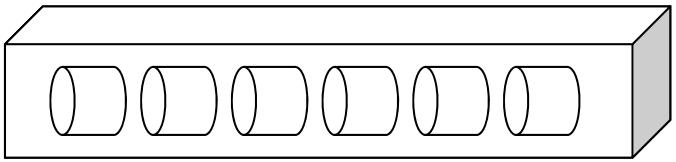
Category	Description
input	Used to read an element from a container. An input iterator can move only in the forward direction (i.e., from the beginning of the container to the end) one element at a time. Input iterators support only one-pass algorithms—the same input iterator cannot be used to pass through a sequence twice.
output	Used to write an element to a container. An output iterator can move only in the forward direction one element at a time. Output iterators support only one-pass algorithms—the same output iterator cannot be used to pass through a sequence twice.
forward	Combines the capabilities of input and output iterators and retains their position in the container (as state information).
bidirectional	Combines the capabilities of a forward iterator with the ability to move in the backward direction (i.e., from the end of the container toward the beginning). Bidirectional iterators support multipass algorithms.
random access	Combines the capabilities of a bidirectional iterator with the ability to directly access any element of the container, i.e., to jump forward or backward by an arbitrary number of elements.

Category hierarchy
(not inheritance hierarchy)



More powerful

Container



Iterator Types Supposed by Containers

Container	Type of iterator supported
Sequence containers (first class)	
`vector`	random access
`deque`	random access
`list`	bidirectional
Associative containers (first class)	
`set`	bidirectional
`multiset`	bidirectional
`map`	bidirectional
`multimap`	bidirectional
Container adapters	
`stack`	no iterators supported
`queue`	no iterators supported
`priority_queue`	no iterators supported

Predefined Iterator typedefs

Predefined typedefs for iterator types	Direction of ++	Capability
iterator	forward	read/write
const_iterator	forward	read
reverse_iterator	backward	read/write
const_reverse_iterator	backward	read

- Not every typedef is defined for every container
- We use const versions of the iterators for traversing read-only containers
- We use reverse iterators to traverse containers in the revsre direction

Iterator Operations (1/2)

Iterator operation	Description
All iterators	
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
Input iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
`p == p1`	Compare iterators for equality.
`p != p1`	Compare iterators for inequality.
Output iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.

- For input and output iterators, it is not possible to save the iterator then used the saved value later

Iterator Operations (2/2)

Iterator operation	Description
Bidirectional iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.
Random-access iterators	
`p += i`	Increment the iterator `p` by `i` positions.
`p -= i`	Decrement the iterator `p` by `i` positions.
`p + i` *or* `i + p`	Expression value is an iterator positioned at `p` incremented by `i` positions.
`p - i`	Expression value is an iterator positioned at `p` decremented by `i` positions.
`p - p1`	Expression value is an integer representing the distance between two elements in the same container.
`p[ i ]`	Return a reference to the element offset from `p` by `i` positions
`p < p1`	Return `true` if iterator `p` is less than iterator `p1` (i.e., iterator `p` is before iterator `p1` in the container); otherwise, return `false`.
`p <= p1`	Return `true` if iterator `p` is less than or equal to iterator `p1` (i.e., iterator `p` is before iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.
`p > p1`	Return `true` if iterator `p` is greater than iterator `p1` (i.e., iterator `p` is after iterator `p1` in the container); otherwise, return `false`.
`p >= p1`	Return `true` if iterator `p` is greater than or equal to iterator `p1` (i.e., iterator `p` is after iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.

Algorithms

- STL algorithms can be used generically across a variety of containers
  - Inserting, deleting, searching, sorting, etc.

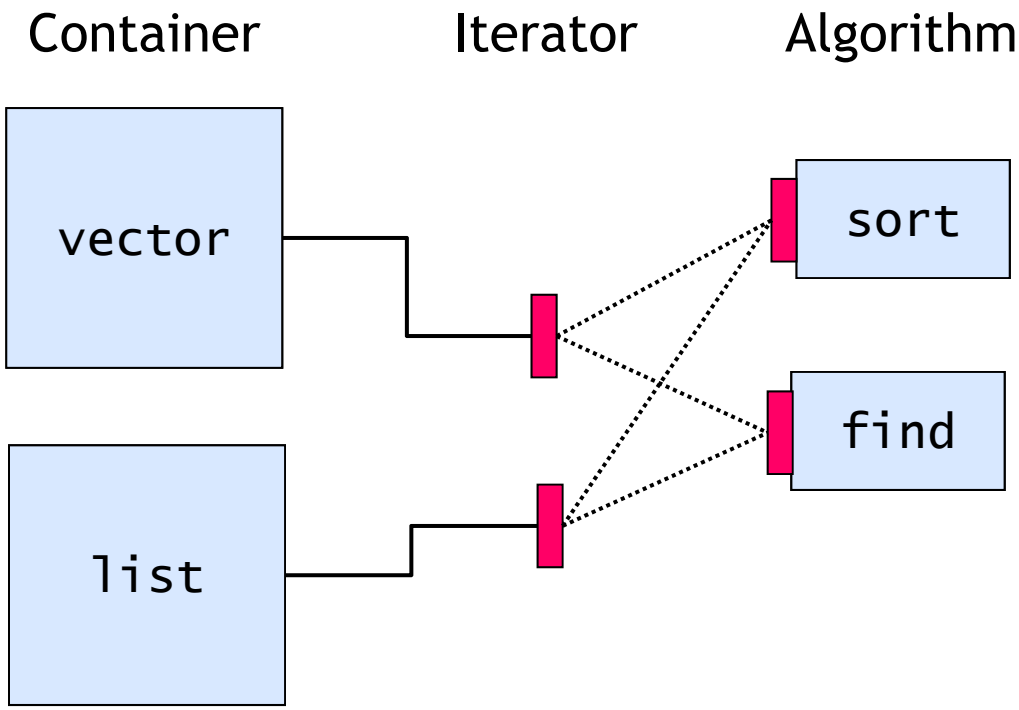
Some STL Algorithms

Nonmodifying sequence algorithms			
adjacent_find	equal	find_end	mismatch
count	find	find_first_of	search
count_if	find_each	find_if	search_n

Mutating-sequence algorithms			
copy	partition	replace_copy	stable_partition
copy_backward	random_shuffle	replace_copy_if	swap
fill	remove	replace_if	swap_ranges
fill_n	remove_copy	reverse	transform
generate	remove_copy_if	reverse_copy	unique
generate_n	remove_if	rotate	unique_copy
iter_swap	replace	rotate_copy	

Containers vs Iterators vs Algorithms

- Iterators allow generic algorithms to 'plug in' to different containers
  - As long as the container supports the iterator that the algorithm uses to access container elements, the algorithm can be applied to it



Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

Sequence Containers: Common Operations

Member function	Description
`front`	Returns a reference to the first element in a non-empty container
`back`	Returns a reference to the last element in a non-empty container
`begin`	Returns an iterator that points to the first element of the container
`end`	Returns an iterator that points to the next position after the last element of the container
`push_back`	Inserts a new element at the end of the container
`pop_back`	Remove the last element of the container
`insert`	Inserts new elements before the element at the specified position
`erase`	Returns an iterator that points to the next position after the last element of the container
`...`	

Sequence Containers: Performance Tips

- Applications that require frequent insertions and deletions at both ends of a container normally use a `deque` rather than a `vector`
  - although we can insert and delete elements at the front and back of both a `vector` and a `deque`
- Applications with frequent insertions and deletions in the middle and/or at the extreme of a container normally use a `list`
  - Due to its efficient implementation of insertion and deletion anywhere in the data structure

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector`, `list`, `deque`)
  - Associative Containers (`map`/`multimap`, `set`/`multiset`)
  - Function Objects
  - Container Adapters (`stack`, `queue`, `priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

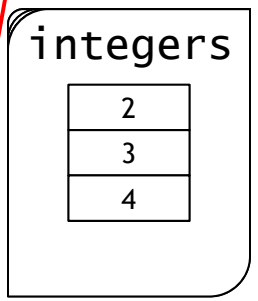
Sequence Containers: `vector`

- An array-based container
- Dynamically resizable, assignable, optional bounds checking (using at)
- Includes functions for insertion at any position of the vector
- Due to the array-ness of the vector, insertion is efficient only at the end

Using vector and iterators (1/2)

```
3  #include <iostream>
4  #include <vector> // vect
5  using namespace std;
6
7  // prototype for function template printVector
8  template < typename T > void printVector( const vector< T > &integers2 );
9
10 int main()
11 {
12    const int SIZE = 6; // define array size
13    int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 }; // initialize array
14    vector< int > integers; // create vector of ints
15
16    cout << "The initial size of integers is: " << integers.size()
17       << "\nThe initial capacity of integers is: " << integers.capacity();
18
19    // function push_back is in every sequence co
20    integers.push_back( 2 );
21    integers.push_back( 3 );
22    integers.push_back( 4 );
23
24    cout << "\nThe size of integers is: " << integers.size()
25       << "\nThe capacity of integers is: " << integers.capacity();
26    cout << "\n\nOutput array using pointer notation: ";
27
28    // display array using pointer notation
29    for ( int *ptr = array; ptr != array + SIZE; ptr++ )
30       cout << *ptr << ' ';
31
```

The capacity is not necessarily equal to the vector size. It can be equal or greater, with the extra space allowing to accommodate for growth without the need to reallocate on each insertion




```
The initial size of integers is: 0
The initial capacity of integers is: 0
The size of integers is: 3
The capacity of integers is: 4

Output array using pointer notation: 1 2 3 4 5 6
```

Using vector and iterators (2/2)

```
32      cout << "\nOutput vector using iterator notation: ";
33      printVector( integers );
34      cout << "\nReversed contents of vector integers: ";
35
36      // two const reverse iterators
37      vector< int >::const_reverse_iterator reverseIterator;
38      vector< int >::const_reverse_iterator tempIterator = integers.rend();
39
40      // display vector in reverse order using reverse_iterator
41      for ( reverseIterator = integers.rbegin();
42         reverseIterator!= tempIterator; ++reverseIterator )
43         cout << *reverseIterator << ' ';
44
45      cout << endl;
46   } // end main
47
48   // function template for outputting vector elements
49   template < typename T > void printVector( const vector< T > &integers2 )
50   {
51      typename vector< T >::const_iterator constIterator; // const_iterator
52
53      // display vector elements using const_iterator
54      for ( constIterator = integers2.begin();
55         constIterator != integers2.end(); ++constIterator )
56         cout << *constIterator << ' ';
57   } // end function printVector
```

integers

2
3
4

Use prefix increment when applied to STL iterators because the prefix increment operator does not return a value that must be stored in a temporary object

```
Output vector using iterator notation: 2 3 4
Reversed contents of vector integers: 4 3 2
```

vector: Using Element-Manipulation Functions (1/2)

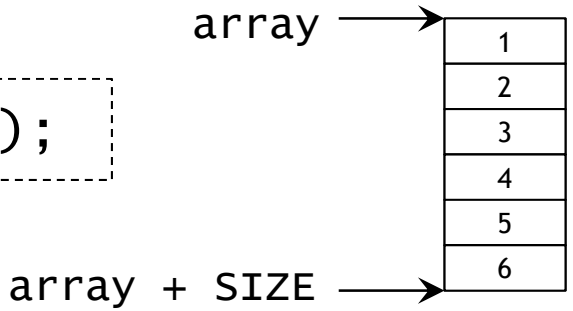
```
 4  #include <iostream>
 5  #include <vector> // vector class-template definition
 6  #include <algorithm> // copy algorithm
 7  #include <iterator> // ostream_iterator iterator
 8  #include <stdexcept> // out_of_range exception
 9  using namespace std;
10
11  int main()
12  {
13     const int SIZE = 6;
14     int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
15     vector< int > integers( array, array + SIZE );
16     ostream_iterator< int > output( cout, " " );
17
18     cout << "Vector integers contains: ";
19     copy( integers.begin(), integers.end(), output );
20
21     cout << "\nFirst element of integers: " << integers.front()
22        << "\nLast element of integers: " << integers.back();
23
24     integers[ 0 ] = 7; // set first element to 7
25     integers.at( 2 ) = 10; // set element at position 2 to 10
26
27     // insert 22 as 2nd element
28     integers.insert( integers.begin() + 1, 22 );
29
30     cout << "\n\nContents of vector integers after changes: ";
31     copy( integers.begin(), integers.end(), output );
```

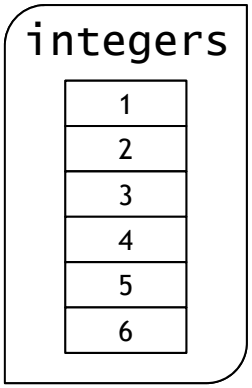
vector: Range Constructor

```
template <class InputIterator>
vector (InputIterator first, InputIterator last,
        const allocator_type& alloc = allocator_type());
```

- Constructs a container with as many elements as the range [first,last), with each element constructed from its corresponding element in that range, in the same order

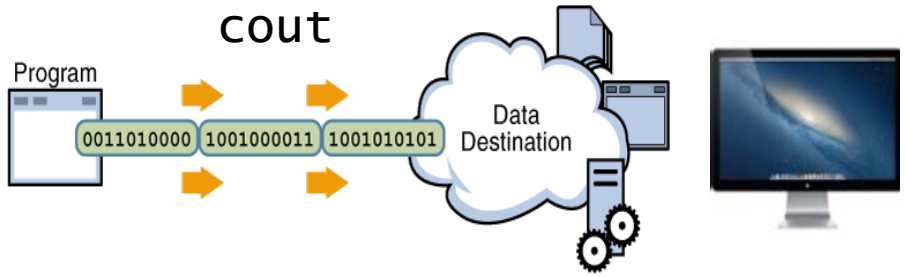
```
vector<int> integers(array, array + SIZE);
```





Class ostream_iterator

- cout (an ostream object/container)



- cout is a container that has a stream of binary values that are output to the display
- ostream_iterator<int> output(cout, " ") is an iterator of an ostream container, which has a stream of integers
  - The first argument of the constructor specifies which ostream container the iterator is associated with
  - The second argument of the constructor specifies the delimiter that is written to the stream after each element is inserted

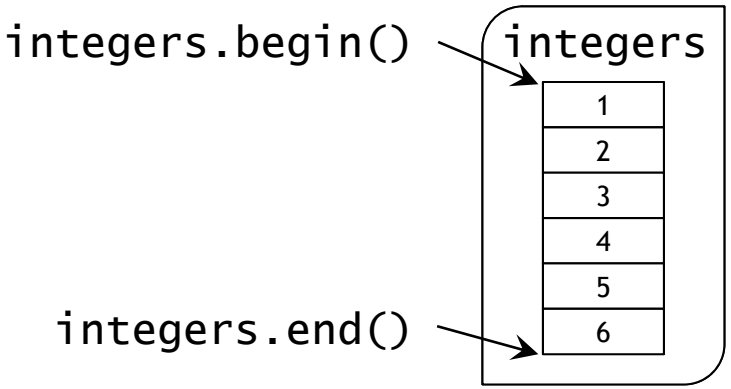
vector: Using Element-Manipulation Functions (1/2)

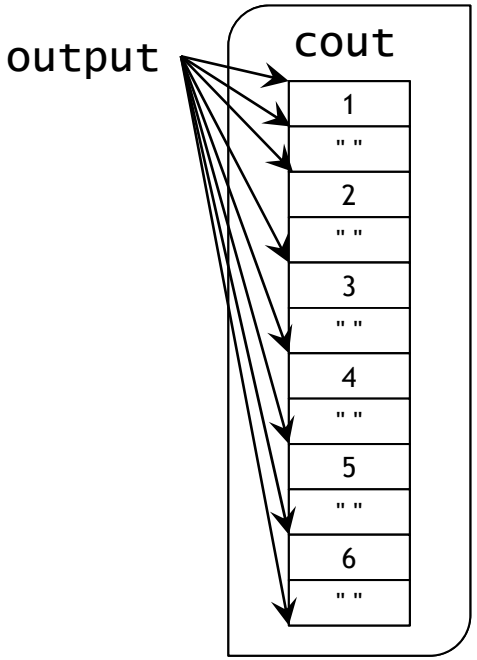
```
4  #include <iostream>
5  #include <vector> // vector class-template definition
6  #include <algorithm> // copy algorithm
7  #include <iterator> // ostream_iterator iterator
8  #include <stdexcept> // out_of_range exception
9  using namespace std;
10
11 int main()
12 {
13    const int SIZE = 6;
14    int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
15    vector< int > integers( array, array + SIZE );
16    ostream_iterator< int > output( cout, " " );
17
18    cout << "Vector integers contains: ";
19    copy( integers.begin(), integers.end(), output );
20
21    cout << "\nFirst element of integers: " << integers.front()
22       << "\nLast element of integers: " << integers.back();
23
24    integers[ 0 ] = 7; // set first element to 7
25    integers.at( 2 ) = 10; // set element at position 2 to 10
26
27    // insert 22 as 2nd element
28    integers.insert( integers.begin() + 1, 22 );
29
30    cout << "\n\nContents of vector integers after changes: ";
31    copy( integers.begin(), integers.end(), output );
```

Outputs only int-compatible values

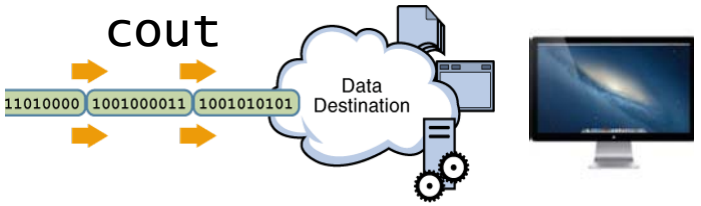
Algorithm copy

- copy(integers.begin(), integers.end(), output)





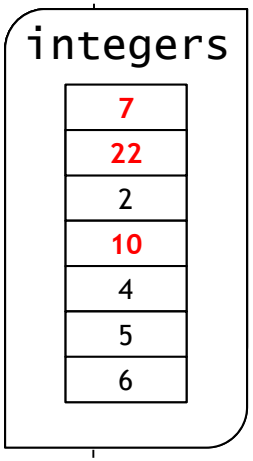

```
template<class InputIterator, class OutputIterator>
  OutputIterator copy (InputIterator first,
  InputIterator last, OutputIterator result)
{
  while (first!=last) {
    *result = *first;
    ++result; ++first;
  }
  return result;
}
```



vector: Using Element-Manipulation Functions (1/2)

```
 4  #include <iostream>
 5  #include <vector> // vector class-template definition
 6  #include <algorithm> // copy algorithm
 7  #include <iterator> // ostream_iterator iterator
 8  #include <stdexcept> // out_of_range exception
 9  using namespace std;
10
11  int main()
12  {
13     const int SIZE = 6;
14     int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
15     vector< int > integers( array, array + SIZE );
16     ostream_iterator< int > output( cout, " " );
17
18     cout << "Vector integers contains: ";
19     copy( integers.begin(), integers.end(), output );
20
21     cout << "\nFirst element of integers: " << integers.front()
22        << "\nLast element of integers: " << integers.back();
23
24     integers[ 0 ] = 7; // set first element to 7
25     integers.at( 2 ) = 10; // set element at position 2 to 10
26
27     // insert 22 as 2nd element
28     integers.insert( integers.begin() + 1, 22 );
29
30     cout << "\n\nContents of vector integers after changes: ";
31     copy( integers.begin(), integers.end(), output );
```

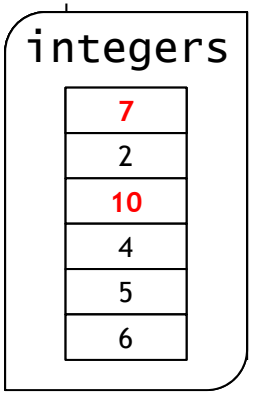
```
Vector integers contains: 1 2 3 4 5 6
First element of integers: 1
Last element of integers: 6

Contents of vector integers after changes: 7 22 2 10 4 5 6
```



vector: Using Element-Manipulation Functions (2/2)

```
33      // access out-of-range element
34      try
35      {
36         integers.at( 100 ) = 777;
37      } // end try
38      catch ( out_of_range &outOfRange ) // out_of_range exception
39      {
40         cout << "\n\nException: " << outOfRange.what();
41      } // end catch
42
43      // erase first element
44      integers.erase( integers.begin() );
45      cout << "\n\nVector integers after erasing first element: ";
46      copy( integers.begin(), integers.end(), output );
47
48      // erase remaining elements
49      integers.erase( integers.begin(), integers.end() );
50      cout << "\nAfter erasing all elements, vector integers "
51         << ( integers.empty() ? "is" : "is not" ) << " empty";
52
53      // insert elements from array
54      integers.insert( integers.begin(), array, array + SIZE );
55      cout << "\n\nContents of vector integers before clear: ";
56      copy( integers.begin(), integers.end(), output );
57
58      // empty integers; clear calls erase to empty a collection
59      integers.clear();
60      cout << "\nAfter clear, vector integers "
61         << ( integers.empty() ? "is" : "is not" ) << " empty" << endl;
62   } // end main
```

```
Exception: invalid vector<T> subscript

Vector integers after erasing first element: 22 2 10 4 5 6
After erasing all elements, vector integers is empty

Contents of vector integers before clear: 1 2 3 4 5 6
After clear, vector integers is empty
```

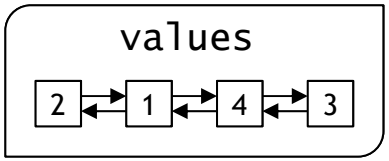


Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector`, `list`, `deque`)
  - Associative Containers (`map/multimap`, `set/multiset`)
  - Function Objects
  - Container Adapters (`stack`, `queue`, `priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

Using `list` (1/4)

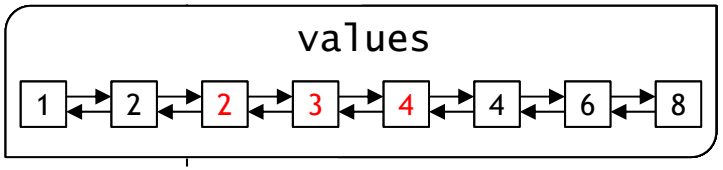
```
 3  #include <iostream>
 4  #include <list> // list class-template definition
 5  #include <algorithm> // copy algorithm
 6  #include <iterator> // ostream_iterator
 7  using namespace std;
 8
 9  // prototype for function template printList
10  template < typename T > void printList( const list< T > &listRef );
11
12  int main()
13  {
14     const int SIZE = 4;
15     int array[ SIZE ] = { 2, 6, 4, 8 };
16     list< int > values; // create list of ints
17     list< int > otherValues; // create list of ints
18
19     // insert items in values
20     values.push_front( 1 );
21     values.push_front( 2 );
22     values.push_back( 4 );
23     values.push_back( 3 );
24
```

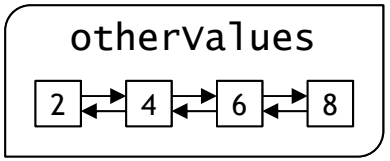


Using list (2/4)

```
25      cout << "values contains: ";
26      printList( values );
27
28      values.sort(); // sort values
29      cout << "\nvalues after sorting contains: ";
30      printList( values );
31
32      // insert elements of array into otherValues
33      otherValues.insert( otherValues.begin(), array, array + SIZE );
34      cout << "\nAfter insert, otherValues contains: ";
35      printList( otherValues );
36
37      // remove otherValues elements and insert at end of values
38      values.splice( values.end(), otherValues );
39      cout << "\nAfter splice, values contains: ";
40      printList( values );
41
42      values.sort(); // sort values
43      cout << "\nAfter sort, values contains: ";
44      printList( values );
45
46      // insert elements of array into otherValues
47      otherValues.insert( otherValues.begin(), array, array + SIZE );
48      otherValues.sort();
```

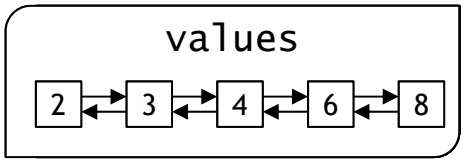
```
int array[ SIZE ] = { 2, 6, 4, 8 };
```

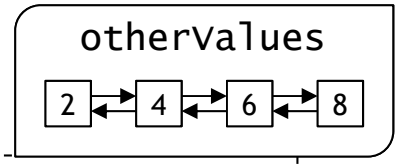





```
93  template < typename T > void printList( const list< T > &listRef )
94  {
95     if ( listRef.empty() ) // list is empty
96        cout << "List is empty";
97     else
98     {
99        ostream_iterator< T > output( cout, " " );
100       copy( listRef.begin(), listRef.end(), output );
101    } // end else
102 } // end function printList
```

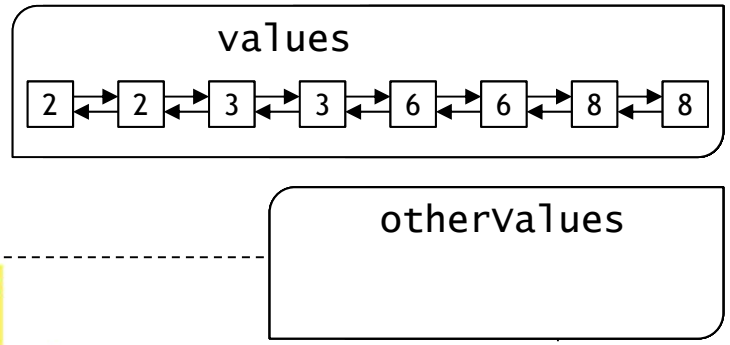
Using list (3/4)






```
49      cout << "\nAfter insert and sort, otherValues contains: ";
50      printList( otherValues );
51
52      // remove otherValues elements and insert into values in sorted order
53      values.merge( otherValues );
54      cout << "\nAfter merge:\n   values contains: ";
55      printList( values );
56      cout << "\n   otherValues contains: ";
57      printList( otherValues );
58
59      values.pop_front(); // remove element from front
60      values.pop_back(); // remove element from back
61      cout << "\nAfter pop_front and pop_back:\n   values contains: "
62      printList( values );
63
64      values.unique(); // remove duplicate elements
65      cout << "\nAfter unique, values contains: ";
66      printList( values );
67
```

Using list (4/4)




```
68      // swap elements of values and otherValues
69      values.swap( otherValues );
70      cout << "\nAfter swap:\n   values contains: ";
71      printList( values );
72      cout << "\n   otherValues contains: ";
73      printList( otherValues );
74
75      // replace contents of values with elements of otherValues
76      values.assign( otherValues.begin(), otherValues.end() );
77      cout << "\nAfter assign, values contains: ";
78      printList( values );
79
80      // remove otherValues elements and insert into values in sorted order
81      values.merge( otherValues );
82      cout << "\nAfter merge, values contains: ";
83      printList( values );
84
85      values.remove( 4 ); // remove all 4s
86      cout << "\nAfter remove( 4 ), values contains: ";
87      printList( values );
88      cout << endl;
89   } // end main
```

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector`, `list`, `deque`)
  - Associative Containers (`map`/`multimap`, `set`/`multiset`)
  - Function Objects
  - Container Adapters (`stack`, `queue`, `priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

Using deque

```
3  #include <iostream>
4  #include <deque> // deque class-template definition
5  #include <algorithm> // copy algorithm
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  int main()
10 {
11    deque< double > values; // create deque of doubles
12    ostream_iterator< double > output( cout, " " );
13
14    // insert elements in values
15    values.push_front( 2.2 );
16    values.push_front( 3.5 );
17    values.push_back( 1.1 );
18
19    cout << "values contains: ";
20
21    // use subscript operator to obtain elements of values
22    for ( unsigned int i = 0; i < values.size(); i++ )
23       cout << values[ i ] << ' ';
24
25    values.pop_front(); // remove first element
26    cout << "\nAfter pop_front, values contains: ";
27    copy( values.begin(), values.end(), output );
28
29    // use subscript operator to modify element at location 1
30    values[ 1 ] = 5.4;
31    cout << "\nAfter values[ 1 ] = 5.4, values contains: ";
32    copy( values.begin(), values.end(), output );
33    cout << endl;
34 } // end main
```

```
values contains: 3.5 2.2 1.1
After pop_front, values contains: 2.2 1.1
After values[ 1 ] = 5.4, values contains: 2.2 5.4
```

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map`/`multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

Associative Containers: map

```
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;
```

Comparison class

- maps are associative containers that store elements formed by a combination of a key value (search key) and a mapped value, following a specific order
  - Indicated by its internal comparison object (of type Compare)
  - One-to-one mapping
  - Also called associative arrays
- In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key
- A unique feature of map among associative containers is that they implement the direct access operator (operator[]), which allows for direct access of the mapped value

Using map (1/2)

```
3  #include <iostream>
4  #include <map> // map class-template definition
5  using namespace std;
6
7  // define short name for map type used in this program
8  typedef map< int, double, less< int > > Mid;
9
10 int main()
11 {
12    Mid pairs;
13
14    // insert eight value_type objects in pairs
15    pairs.insert( Mid::value_type( 15, 2.7 ) );
16    pairs.insert( Mid::value_type( 30, 111.11 ) );
17    pairs.insert( Mid::value_type( 5, 1010.1 ) );
18    pairs.insert( Mid::value_type( 10, 22.22 ) );
19    pairs.insert( Mid::value_type( 25, 33.333 ) );
20    pairs.insert( Mid::value_type( 5, 77.54 ) ); // dup ignored
21    pairs.insert( Mid::value_type( 20, 9.345 ) );
22    pairs.insert( Mid::value_type( 15, 99.3 ) ); // dup ignored
23
```

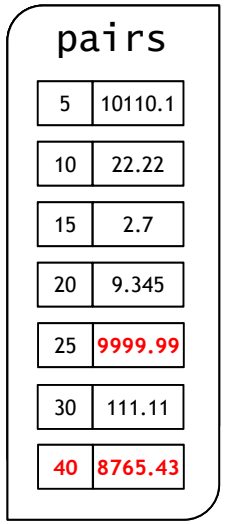
Makes code with long type names easier to read

pairs

5	10110.1
10	22.22
15	2.7
20	9.345
25	33.333
30	111.11

Using map (2/2)

```
24      cout << "pairs contains:\nKey\tValue\n";
25
26      // use const_iterator to walk through elements of pairs
27      for ( Mid::const_iterator iter = pairs.begin();
28         iter != pairs.end(); ++iter )
29         cout << iter->first << '\t' << iter->second << '\n';
30
31      pairs[ 25 ] = 9999.99; // use subscripting to change value for key 25
32      pairs[ 40 ] = 8765.43; // use subscripting to insert value for key 40
33
34      cout << "\nAfter subscript operations, pairs contains:\nKey\tValue\n";
35
36      // use const_iterator to walk through elements of pairs
37      for ( Mid::const_iterator iter2 = pairs.begin();
38         iter2 != pairs.end(); ++iter2 )
39         cout << iter2->first << '\t' << iter2->second << '\n';
40
41      cout << endl;
42   } // end main
```




```
pairs contains:
Key     Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      33.333
30      111.11

After subscript operations, pairs contains:
Key     Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      9999.99
30      111.11
40      8765.43
```

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

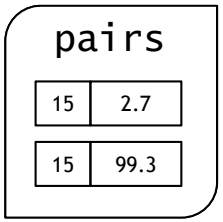
Associative Containers: multimap

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

- multimaps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order, and where multiple elements can have equivalent values

Using multimap (1/2)

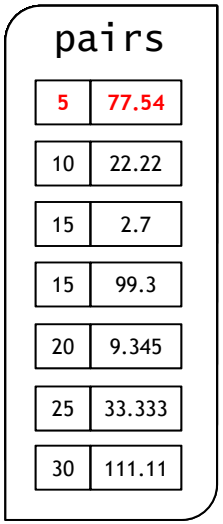
```
3  #include <iostream>
4  #include <map> // multimap class-template definition
5  using namespace std;
6
7  // define short name for multimap type used in this program
8  typedef multimap< int, double, less< int > > Mmid;
9
10 int main()
11 {
12    Mmid pairs; // declare the multimap pairs
13
14    cout << "There are currently " << pairs.count( 15 )
15       << " pairs with key 15 in the multimap\n";
16
17    // insert two value_type objects in pairs
18    pairs.insert( Mmid::value_type( 15, 2.7 ) );
19    pairs.insert( Mmid::value_type( 15, 99.3 ) );
20
21    cout << "After inserts, there are " << pairs.count( 15 )
22       << " pairs with key 15\n\n";
23
```




```
There are currently 0 pairs with key 15 in the multimap
After inserts, there are 2 pairs with key 15
```

Using multimap (2/2)

```
24      // insert five value_type objects in pairs
25      pairs.insert( Mmid::value_type( 30, 111.11 ) );
26      pairs.insert( Mmid::value_type( 10, 22.22 ) );
27      pairs.insert( Mmid::value_type( 25, 33.333 ) );
28      pairs.insert( Mmid::value_type( 20, 9.345 ) );
29      pairs.insert( Mmid::value_type( 5, 77.54 ) );
30
31      cout << "Multimap pairs contains:\nKey\tValue\n";
32
33      // use const_iterator to walk through elements of pairs
34      for ( Mmid::const_iterator iter = pairs.begin();
35         iter != pairs.end(); ++iter )
36         cout << iter->first << '\t' << iter->second << '\n';
37
38      cout << endl;
39   } // end main
```




```
Multimap pairs contains:
Key     Value
5       77.54
10      22.22
15      2.7
15      99.3
20      9.345
25      33.333
30      111.11
```

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

Associative Containers: set

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

- sets are containers that store unique elements following a specific order
  - Indicated by its internal comparison object (of type Compare)
- In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique
- The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container

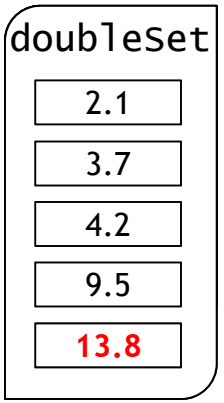
Using set (1/2)

```
3  #include <iostream>
4  #include <set>
5  #include <algorithm>
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  // define short name for set type used in this program
10 typedef set< double, less< double > > DoubleSet;
11
12 int main()
13 {
14    const int SIZE = 5;
15    double a[ SIZE ] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
16    DoubleSet doubleSet( a, a + SIZE );
17    ostream_iterator< double > output( cout, " " );
18
19    cout << "doubleSet contains: ";
20    copy( doubleSet.begin(), doubleSet.end(), output );
```


```
doubleSet contains: 2.1 3.7 4.2 9.5
```

Using set (2/2)

```
22      // p represents pair containing const_iterator and bool
23      pair< DoubleSet::const_iterator, bool > p;
24
25      // insert 13.8 in doubleSet; insert returns pair in which
26      // p.first represents location of 13.8 in doubleSet and
27      // p.second represents whether 13.8 was inserted
28      p = doubleSet.insert( 13.8 ); // value not in set
29      cout << "\n\n" << *( p.first )
30         << ( p.second ? " was" : " was not" ) << " inserted";
31      cout << "\ndoubleSet contains: ";
32      copy( doubleSet.begin(), doubleSet.end(), output );
33
34      // insert 9.5 in doubleSet
35      p = doubleSet.insert( 9.5 ); // value already in set
36      cout << "\n\n" << *( p.first )
37         << ( p.second ? " was" : " was not" ) << " inserted";
38      cout << "\ndoubleSet contains: ";
39      copy( doubleSet.begin(), doubleSet.end(), output );
40      cout << endl;
41   } // end main
```




```
13.8 was inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8

9.5 was not inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8
```

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

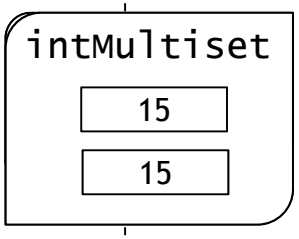
Associative Containers: multiset

```
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;
```

- multisets are containers that store elements following a specific order, and where multiple elements can have equivalent values

Using multiset (1/3)

```
3  #include <iostream>
4  #include <set> // multiset class-template definition
5  #include <algorithm> // copy algorithm
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  // define short name for multiset type used in this program
10 typedef multiset< int, less< int > > Ims;
11
12 int main()
13 {
14    const int SIZE = 10;
15    int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
16    Ims intMultiset; // Ims is typedef for "integer multiset"
17    ostream_iterator< int > output( cout, " " );
18
19    cout << "There are currently " << intMultiset.count( 15 )
20       << " values of 15 in the multiset\n";
21
22    intMultiset.insert( 15 ); // insert 15 in intMultiset
23    intMultiset.insert( 15 ); // insert 15 in intMultiset
24    cout << "After inserts, there are " << intMultiset.count( 15 )
25       << " values of 15 in the multiset\n\n";
```

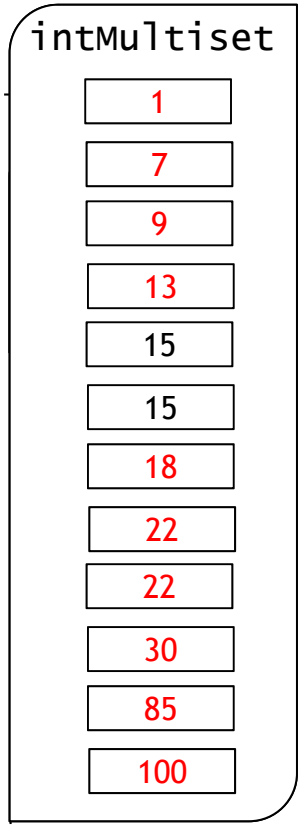



```
There are currently 0 values of 15 in the multiset
After inserts, there are 2 values of 15 in the multiset
```

Using multiset (2/3)

```
int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
```

```
27      // iterator that cannot be used to change element values
28      Ims::const_iterator result;
29
30      // find 15 in intMultiset; find returns iterator
31      result = intMultiset.find( 15 );
32
33      if ( result != intMultiset.end() ) // if iterator not at end
34         cout << "Found value 15\n"; // found search value 15
35
36      // find 20 in intMultiset; find returns iterator
37      result = intMultiset.find( 20 );
38
39      if ( result == intMultiset.end() ) // will be true hence
40         cout << "Did not find value 20\n"; // did not find 20
41
42      // insert elements of array a into intMultiset
43      intMultiset.insert( a, a + SIZE );
44      cout << "\nAfter insert, intMultiset contains:\n";
45      copy( intMultiset.begin(), intMultiset.end(), output );
```



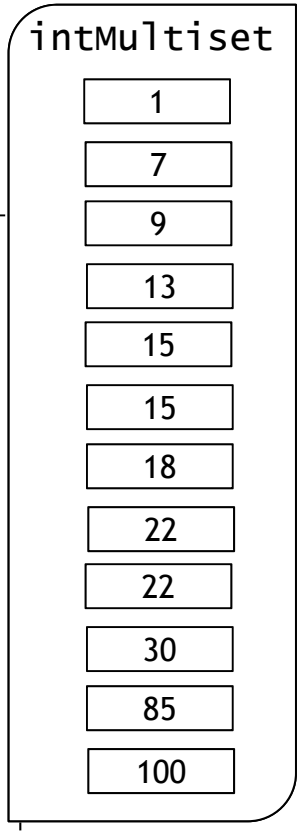

```
Found value 15
Did not find value 20

After insert, intMultiset contains:
1 7 9 13 15 15 18 22 22 30 85 100
```

Using multiset (3/3)

The earliest occurrence of the value 22

The element after the last occurrence of the value 22

```
46
47       // determine lower and upper bound of 22 in intMultiset
48       cout << "\n\nLower bound of 22: "
49          << *( intMultiset.lower_bound( 22 ) );
50       cout << "\nUpper bound of 22: " << *( intMultiset.upper_bound( 22 ) );
51
52       // p represents pair of const_iterators
53       pair< Ims::const_iterator, Ims::const_iterator > p;
54
55       // use equal_range to determine lower and upper bound
56       // of 22 in intMultiset
57       p = intMultiset.equal_range( 22 );
58
59       cout << "\n\nequal_range of 22:" << "\n   Lower bound: "
60          << *( p.first ) << "\n   Upper bound: " << *( p.second );
61       cout << endl;
62    } // end main
```




```
Lower bound of 22: 22
Upper bound of 22: 30

equal_range of 22:
   Lower bound: 22
   Upper bound: 30
```

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

Function Objects (Functors)

- Objects that represent functions
- Overrides operator()
- Functors are usually templated

```
template <class T>
struct less : public binary_function<T, T, bool> {
  bool operator() (const T &x, const T &y) const {
    return (x < y);
  }
}
```

This will only work if < is defined for T

Function Objects in Standard Library

- Defined in <functional>

STL function objects	Type	STL function objects	Type
divides< T >	arithmetic	logical_or< T >	logical
equal_to< T >	relational	minus< T >	arithmetic
greater< T >	relational	modulus< T >	arithmetic
greater_equal< T >	relational	negate< T >	arithmetic
less< T >	relational	not_equal_to< T >	relational
less_equal< T >	relational	plus< T >	arithmetic
logical_and< T >	logical	multiplies< T >	arithmetic
logical_not< T >	logical		

Writing Our Own Function Objects: An Example

```
#include <iostream>
#include <vector>
#include <algorithm> // std::for_each
using namespace std;

void myFunction (const int &x) {  // function
    cout << x * 2 << endl;
}

class myFunctorClass { // function object type
private:
    int multiplier;
public:
    myFunctorClass(const int &mul) : multiplier(mul) {}
    void operator() (const int &x) {
        cout << x * multiplier << endl;
    }
};

int main () {
    int sequence[5] = {1,2,3,4,5};
    vector<int> v(sequence, sequence+5);
                                    // myFunction<2>
    for_each(v.begin(), v.end(), myFunction); // Apply function

    myFunctorClass myFunctor(2); // function object
    for_each(v.begin(), v.end(), myFunctor); // Apply function

    return 0;
}
```

```
template<class InputIterator, class Function>
Function for_each(InputIterator first,
InputIterator last, Function fn)
{
  while (first!=last) {
    fn (*first);
    ++first;
  }
  return fn;
}
```

```
template <int multiplier>
void myFunction (const int &x) {
    cout << x * multiplier << endl;
}
```

```
2
4
6
8
10
2
4
6
8
10
```

Functors and Algorithms

- Note that generic algorithms do not care if the operation argument is
  - a function,
  - a pointer to a function, or
  - a functor

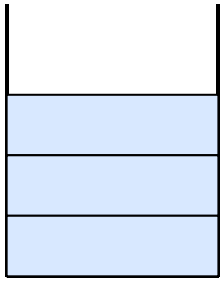
Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

Container Adapters: stack

```
template < class T, class Container = deque<T> >
class stack;
```

- T: Type of the elements
- Container: Type of the underlying container object used to store and access the elements
- stacks are container adapters specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from the end of the container



Stack

Using stack (1/2)

```
3  #include <iostream>
4  #include <stack> // stack adapter definition
5  #include <vector> // vector class-template definition
6  #include <list> // list class-template definition
7  using namespace std;
8
9  // pushElements function-template prototype
10 template< typename T > void pushElements( T &stackRef );
11
12 // popElements function-template prototype
13 template< typename T > void popElements( T &stackRef );
14
15 int main()
16 {
17    // stack with default underlying deque
18    stack< int > intDequeStack;
19
20    // stack with underlying vector
21    stack< int, vector< int > > intVectorStack;
22
23    // stack with underlying list
24    stack< int, list< int > > intListStack;
```

Best performance

Using stack (2/2)

```
26      // push the values 0-9 onto each stack
27      cout << "Pushing onto intDequeStack: ";
28      pushElements( intDequeStack );
29      cout << "\nPushing onto intVectorStack: ";
30      pushElements( intVectorStack );
31      cout << "\nPushing onto intListStack: ";
32      pushElements( intListStack );
33      cout << endl << endl;
34
35      // display and remove elements from each stack
36      cout << "Popping from intDequeStack: ";
37      popElements( intDequeStack );
38      cout << "\nPopping from intVectorStack: ";
39      popElements( intVectorStack );
40      cout << "\nPopping from intListStack: ";
41      popElements( intListStack );
42      cout << endl;
43   } // end main
44
45   // push elements onto stack object to which stackRef refers
46   template< typename T > void pushElements( T &stackRef )
47   {
48      for ( int i = 0; i < 10; i++ )
49      {
50         stackRef.push( i ); // push element onto stack
51         cout << stackRef.top() << ' '; // view (and display) top element
52      } // end for
53   } // end function pushElements
54
55   // pop elements from stack object to which stackRef refers
56   template< typename T > void popElements( T &stackRef )
57   {
58      while ( !stackRef.empty() )
59      {
60         cout << stackRef.top() << ' '; // view (and display) top element
61         stackRef.pop(); // remove top element
62      } // end while
63   } // end function popElements
```

```
Pushing onto intDequeStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intVectorStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intListStack: 0 1 2 3 4 5 6 7 8 9

Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0
```

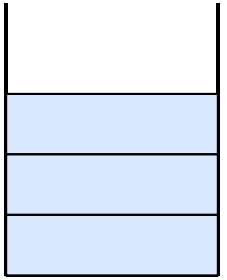
Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

Container Adapters: queue

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

- `T`: Type of the elements
- `Container`: Type of the underlying container object used to store and access the elements
- queues are container adapters specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other



Queue

Using queue

```
 3  #include <iostream>
 4  #include <queue> // queue adapter definition
 5  using namespace std;
 6
 7  int main()
 8  {
 9     queue< double > values; // queue with doubles
10
11     // push elements onto queue values
12     values.push( 3.2 );
13     values.push( 9.8 );
14     values.push( 5.4 );
15
16     cout << "Popping from values: ";
17
18     // pop elements from queue
19     while ( !values.empty() )
20     {
21        cout << values.front() << ' '; // view front element
22        values.pop(); // remove element
23     } // end while
24
25     cout << endl;
26  } // end main
```

```
Popping from values: 3.2 9.8 5.4
```

Outline

- Introduction to Templates
  - Function Templates
  - Class Templates
  - Default Template Types & Non-type Template Parameters
- Standard Template Library
  - Introduction to STL
  - Sequence Containers (`vector, list, deque`)
  - Associative Containers (`map/multimap, set/multiset`)
  - Function Objects
  - Container Adapters (`stack, queue, priority_queue`)
  - Algorithms
  - Near Containers (`bitset`)

Container Adapters: priority_ueue

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

- T: Type of the elements
- Container: Type of the underlying container object used to store and access the elements
- Compare: Comparison class
- priority_queues are container adapters specifically designed such that its first element is always the greatest of the elements it contains, according to some strict weak ordering condition

Using priority_queue

```
3  #include <iostream>
4  #include <queue> // priority_queue adapter definition
5  using namespace std;
6
7  int main()
8  {
9     priority_queue< double > priorities; // create priority_queue
10
11    // push elements onto priorities
12    priorities.push( 3.2 );
13    priorities.push( 9.8 );
14    priorities.push( 5.4 );
15
16    cout << "Popping from priorities: ";
17
18    // pop element from priority_queue
19    while ( !priorities.empty() )
20    {
21       cout << priorities.top() << ' '; // view top element
22       priorities.pop(); // remove top element
23    } // end while
24
25    cout << endl;
26 } // end main
```

```
Popping from priorities: 9.8 5.4 3.2
```