The Standard Template Library

STL in C++

- **The Standard Template Library (STL)**: an extensive library of generic templates for classes and functions.
- Categories of Templates:
  - **Containers**: Class templates for objects that store and organize data
  - **Iterators**: Class templates for objects that behave like pointers, and are used to access the individual data elements in a container
  - **Algorithms**: Function templates that perform various operations on elements of containers

Containers

- **Sequence Containers**
  - Stores data sequentially in memory, in a fashion similar to an array
- **Associative Containers**
  - Stores data in a non-sequential way that makes it faster to locate elements

Table 17-1 Sequence Containers

Container Class	Description
`array`	A fixed-size container that is similar to an array
`deque`	A double-ended queue. Like a `vector`, but designed so that values can be quickly added to or removed from the front and back. (This container will be discussed in Chapter 19.)
`forward_list`	A singly linked list of data elements. Values may be inserted to or removed from any position. (This container will be discussed in Chapter 18.)
`list`	A doubly linked list of data elements. Values may be inserted to or removed from any position. (This container will be discussed in Chapter 18.)
`vector`	A container that works like an expandable array. Values may be added to or removed from a `vector`. The `vector` automatically adjusts its size to accommodate the number of elements it contains.

Containers

Table 17-2 Associative Containers

Container Class	Description
`set`	Stores a set of unique values that are sorted. No duplicates are allowed.
`multiset`	Stores a set of unique values that are sorted. Duplicates are allowed.
`map`	Maps a set of keys to data elements. Only one key per data element is allowed. Duplicates are not allowed. The elements are sorted in order of their keys.
`multimap`	Maps a set of keys to data elements. Many keys per data element are allowed. Duplicates are allowed. The elements are sorted in order of their keys.
`unordered_set`	Like a `set`, except that the elements are not sorted
`unordered_multiset`	Like a `multiset`, except that the elements are not sorted
`unordered_map`	Like a `map`, except that the elements are not sorted
`unordered_multimap`	Like a `multimap`, except that the elements are not sorted

Container Adapters

Table 17-3 Container Adapter Classes

Container Adapter Class	Description
`stack`	An adapter class that stores elements in a `deque` (by default). A `stack` is a last-in, first-out (LIFO) container. When you retrieve an element from a `stack`, the `stack` always gives you the last element that was inserted. (This class will be discussed in Chapter 19.)
`queue`	An adapter class that stores elements in a `deque` (by default). A `queue` is a first-in, first-out (FIFO) container. When you retrieve an element from a `stack`, the `stack` always gives you the first, or earliest, element that was inserted. (This class will be discussed in Chapter 19.)
`priority_queue`	An adapter class that stores elements in a `vector` (by default). A data structure in which the element that you retrieve is always the element with the greatest value. (This class will be discussed in Chapter 19.)

STL Header Files

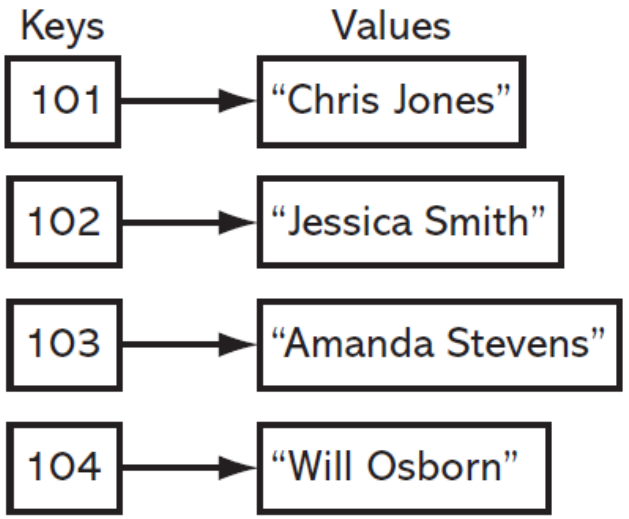
Table 17-4 Header Files

Header File	Classes
<array>	array
<deque>	deque
<forward_list>	forward_list
<list>	list
<map>	map, multimap
<queue>	queue, priority_queue
<set>	set, multiset
<stack>	stack
<unordered_map>	unordered_map, unordered_multimap
<unordered_set>	unordered_set, unordered_multiset
<vector>	vector

The `map`, `multimap`, and `unordered_map` Classes

Maps general concept

- A *map* is an associative container
- Each element that is stored in a map has two parts: a *key* and a *value*.
- To retrieve a specific value from a map, you use the key that is associated with that value.
- This is similar to the process of looking up a word in the dictionary, where the words are keys and the definitions are values.
- Example: a map in which employee IDs are the keys and employee names are the values.
- You use an employee's ID to look up that employee's name.



The map class

- You can use the STL `map` class to store key-value pairs.
- The keys that are stored in a `map` container are unique – no duplicates.
- The `map` class is declared in the `<map>` header file.

Default Constructor	`map<keyDataType, valueDataType> name;` **Creates an empty `map`.**

Range Constructor	`map<keyDataType, valueDataType>`
	`name(iterator1, iterator2);`
	Creates a `map` that is initialized with a range of values from another `map`. *`iterator1`* marks the beginning of the range and *`iterator2`* marks the end.

Copy Constructor	`map<keyDataType, valueDataType> name(map2);` Creates a `map` that is a copy of *`map2`*.

The map class

- Example: defining a `map` container to hold employee ID numbers (as `ints`) and their corresponding employee names (as `strings`):

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


```
map<int, string> employees =
{
   {101, "Chris Jones"}, {102, "Jessica Smith"},
   {103, "Amanda Stevens"}, {104, "Will Osborn"}
};
```

- In the first element, the key is 101 and the value is "Chris Jones".
- In the second element, the key is 102 and the value is "Jessica Smith".
- In the third element, the key is 103 and the value is "Amanda Stevens".
- In the fourth element, the key is 104 and the value is "Will Osborn".

The overloaded [ ] operator

- You can use the [ ] operator to add new elements to a `map`.
- General format:

```
mapName[key] = value;
```

- This adds the key-value pair to the `map`.

- If the key already exists in the `map`, it's associated value will be changed to *`value`*.

```
map<int, string> employees;
employees[110] = "Beth Young";
employees[111] = "Jake Brown";
employees[112] = "Emily Davis";
```

- After this code executes, the `employees` map will contain the following elements:
  - Key = 110, Value = "Beth Young"
  - Key = 111, Value = "Jake Brown"
  - Key = 112, Value = "Emily Davis"

The pair type

- Internally, the elements of a map are stored as instances of the pair type.
- pair is a struct that has two member variables: first and second.
- The element's key is stored in first, and the element's value is stored in second.
- The pair struct is declared in the <utility> header file. When you #include the <map> header file, <utility> is automatically included as well.

Inserting an element using the insert function

- The `map` class provides an `insert()` member function that adds a `pair` object as an element to the `map`.
- You can use the STL function template `make_pair` to construct a `pair` object.
- The `make_pair` function template is declared in the `<utility>` header file.

```
map<int, string> employees;
employees.insert(make_pair(110, "Beth Young"));
employees.insert(make_pair(111, "Jake Brown"));
employees.insert(make_pair(112, "Emily Davis"));
```

- After this code executes, the `employees` map will contain the following elements:
  - Key = 110, Value = "Beth Young"
  - Key = 111, Value = "Jake Brown"
  - Key = 112, Value = "Emily Davis"

Note: If the element that you are inserting with the `insert()` member function has the same key as an existing element, the function will *not* insert the new element.

retrieving an element from a map

- You can use the `at()` member function to retrieve a `map` element by its key:

```
// Create a map containing employee IDs and names.
map<int, string> employees = {
   {101, "Chris Jones"}, {102, "Jessica Smith"},
   {103, "Amanda Stevens"}, {104, "Will Osborn"}
};
// Retrieve a value from the map.
cout << employees.at(103) << endl;
```

Displays "Amanda Stevens"

- To prevent the `at()` member function from throwing an exception (if the specified key does not exist), use the `count` member function to determine whether it exists:

```
// Create a map containing employee IDs and names.
map<int, string> employees =
{
   {101, "Chris Jones"}, {102, "Jessica Smith"},
   {103, "Amanda Stevens"}, {104, "Will Osborn"}
};

// Retrieve a value from the map.
if (employees.count(103))
   cout << employees.at(103) << endl;
else
   cout << "Employee not found.\n";
```

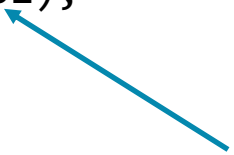
The `count()` member function returns 1 if the specified key exists, or 0 otherwise.

Deleting an element from a map

- You can use the `erase()` member function to retrieve a `map` element by its key:

```
// Create a map containing employee IDs and names.
map<int, string> employees =
{
   {101, "Chris Jones"}, {102, "Jessica Smith"},
   {103, "Amanda Stevens"}, {104, "Will Osborn"}
};

// Delete the employee with ID 102.
employees.erase(102);
```



Deletes Jessica Smith from the map

Range loops and maps

```
// Create a map containing employee IDs and names.
map<int, string> employees =
{
   {101, "Chris Jones"}, {102, "Jessica Smith"},
   {103, "Amanda Stevens"}, {104, "Will Osborn"}
};

// Display each element.
for (pair<int, string> element : employees)
{
   cout << "ID: " << element.first << "\tName: "
        << element.second << endl;
}
```

Remember, each element is a pair.

```
for (auto element : employees)
{
   cout << "ID: " << element.first << "\tName: "
        << element.second << endl;
}
```

auto simplifies this

Iterators and maps

- The `begin()` and `end()` member functions return a bidirectional iterator of the `iterator` type
- The `cbegin()` and `cend()` member functions return a bidirectional iterator of the `const_iterator` type
- The `rbegin()` and `rend()` member functions return a reverse bidirectional iterator of the `reverse_iterator` type
- The `crbegin()` and `crend()` member functions return a reverse bidirectional iterator of the `const_reverse_iterator` type

Iterators and maps

- When an iterator points to a `map` element, it points to an instance of the `pair` type.
- The element has two member variables: `first` and `second`.
- The element's key is stored in `first`, and the element's value is stored in `second`.

Program 17-11

```
1  // This program demonstrates an iterator with a map.
2  #include <iostream>
3  #include <string>
4  #include <map>
5  using namespace std;
6
7  int main()
8  {
9       // Create a map containing employee IDs and names.
10      map<int, string> employees =
11         { {101,"Chris Jones"}, {102,"Jessica Smith"},
12           {103,"Amanda Stevens"},{104,"Will Osborn"} };
13
14      // Create an iterator.
15      map<int, string>::iterator iter;
16
17      // Use the iterator to display each element in the map.
18      for (iter = employees.begin(); iter != employees.end(); iter++)
19      {
20           cout << "ID: " << iter->first
21                << "\tName: " << iter->second << endl;
22      }
23
24      return 0;
25 }
```

Program Output

```
ID: 101 Name: Chris Jones
ID: 102 Name: Jessica Smith
ID: 103 Name: Amanda Stevens
ID: 104 Name: Will Osborn
```

Storing Objects Of Your Own Classes as *Values* in a `map`

- If you want to store an object as a value in a map, there is one requirement for that object's class:
  - It must have a default constructor.
- Consider the following `Contact` class…

```
1  #ifndef CONTACT_H
2  #define CONTACT_H
3  #include <string>
4  using namespace std;
5
6  class Contact
7  {
8  private:
9      string name;
10     string email;
11 public:
12     Contact()          <-- Default constructor
13     {  name = "";
14        email = ""; }
15
16     Contact(string n, string em)
17     {  name = n;
18        email = em; }
19
20     void setName(string n)
21     {  name = n; }
22
23     void setEmail(string em)
24     {  email = em; }
25
26     string getName() const
27     {  return name; }
28
29     string getEmail() const
30     {  return email; }
31 };
32 #endif
```

Storing Objects Of Your Own Classes as *Values* in a `map`

Program 17-14

```
1  #include <iostream>
2  #include <string>
3  #include <map>
4  #include "Contact.h"
5  using namespace std;
6
7  int main()
8  {
9       string searchName;    // The name to search for
10
11      // Create some Contact objects
12      Contact contact1("Ashley Miller", "amiller@faber.edu");
13      Contact contact2("Jacob Brown", "jbrown@gotham.edu");
14      Contact contact3("Emily Ramirez", "eramirez@coolidge.edu");
15
16      // Create a map to hold the Contact objects.
17      map<string, Contact> contacts;
18
19      // Create an iterator for the map.
20      map<string, Contact>::iterator iter;
21
22      // Add the contact objects to the map.
23      contacts[contact1.getName()] = contact1;
24      contacts[contact2.getName()] = contact2;
25      contacts[contact3.getName()] = contact3;
```

In the map, the keys are the contact names, and the values are the `Contact` objects.

Continued…

Storing Objects Of Your Own Classes as *Values* in a map

```
26
27      // Get the name to search for.
28      cout << "Enter a name: ";
29      getline(cin, searchName);
30
31      // Search for the name.
32      iter = contacts.find(searchName);
33
34      // Display the results.
35      if (iter != contacts.end())
36      {
37          cout << "Name: " << iter->second.getName() << endl;
38          cout << "Email: " << iter->second.getEmail() << endl;
39      }
40      else
41      {
42          cout << "Contact not found.\n";
43      }
44
45      return 0;
46  }
```

Program Output (with Example Input Shown in Bold)

```
Enter a name: Emily Ramirez Enter
Name: Emily Ramirez
Email: eramirez@coolidge.edu
```

Program Output (with Example Input Shown in Bold)

```
Enter a name: Billy Clark Enter
Contact not found.
```

Storing Objects Of Your Own Classes as *key* in a map

- If you want to store an object as a key in a map, there is one requirement for that object's class:

  It must overload the < operator.

- Consider the following `Customer` class…

```
1   #ifndef CUSTOMER_H
2   #define CUSTOMER_H
3   #include<string>
4   using namespace std;
5
6   class Customer
7   {
8   private:
9        int custNumber;
10       string name;
11  public:
12       Customer(int cn, string n)
13       {  custNumber = cn;
14          name = n; }
15
16       void setCustNumber(int cn)
17       {  custNumber = cn; }
18
19       void setName(string n)
20       {  name = n; }
21
22       int getCustNumber() const
23       {  return custNumber; }
24
25       string getName() const
26       {  return name; }
27
28       bool operator < (const Customer &right) const
29       {  bool status = false;
30
31          if (custNumber < right.custNumber)
32             status = true;
33
34          return status; }
35  };
36  #endif
```

Storing Objects Of Your Own Classes as *key* in a map

Program 17-17

```
1  #include <iostream>
2  #include <string>
3  #include <map>
4  #include "Customer.h"
5  using namespace std;
6
7  int main()
8  {
9     // Create some Customer objects.
10    Customer customer1(1001, "Sarah Scott");
11    Customer customer2(1002, "Austin Hill");
12    Customer customer3(1003, "Megan Cruz");
13
14    // Create a map to hold the seat assignments.
15    map<Customer, string> assignments;
16
17    // Use the map to store the seat assignments.
18    assignments[customer1] = "1A";
19    assignments[customer2] = "2B";
20    assignments[customer3] = "3C";
21
22    // Display all objects in the map.
23    for (auto element : assignments)
24    {
25       cout << element.first.getName() << "\t"
26            << element.second << endl;
27    }
28
29    return 0;
30 }
```

Program Output

```
Sarah Scott     1A
Austin Hill     2B
Megan Cruz      3C
```

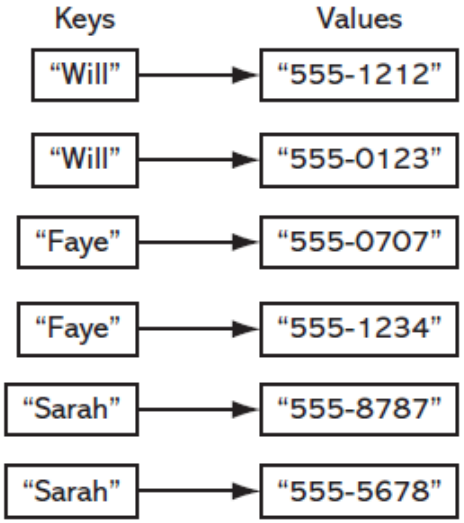
This program assigns seats in a theater to customers. The map uses Customer objects as keys, and seat numbers as values.

Unordered_map class

- The `unordered_map` class is similar to the `map` class, except in two regards:
  - The keys in an `unordered_map` are not sorted
  - The `unordered_map` class has better performance
- You should use the `unordered_map` class instead of the map class if:
  - You will be making a lot of searches on a large number of elements
  - You are not concerned with retrieving them in key order
- The `unordered_map` class is declared in the `<unordered_map>` header file

multimap class

- The `mulitmap` class is a map that allows duplicate keys
- The `mulitmap` class has most of the same member functions as the map class (see Table 17-11 in your textbook)
- The `multimap` class is declared in the `<map>` header file
- Consider a phonebook application where the key is a person's name and the value is that person's phone number.
- A `multimap` container would allow each person to have multiple phone numbers



multimap class

Program 17-19

```
 1  #include <iostream>
 2  #include <string>
 3  #include <map>
 4  using namespace std;
 5
 6  int main()
 7  {
 8     // Define a phonebook multimap.
 9     multimap<string, string> phonebook =
10        { {"Will", "555-1212"},  {"Will", "555-0123"},
11          {"Faye", "555-0707"},  {"Faye", "555-1234"},
12          {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };
13
14     // Display the elements in the multimap.
15     for (auto element : phonebook)
16     {
17        cout << element.first << "\t"
18             << element.second << endl;
19     }
20     return 0;
21  }
```

Program Output

```
Faye    555-0707
Faye    555-1234
Sarah   555-8787
Sarah   555-5678
Will    555-1212
Will    555-0123
```

Adding an element to a multimap

- The `multimap` class does not overload the `[]` operator.
  - So, you cannot use an assignment statement to add a new element to a `multimap`.

- Instead, you will use either the **`emplace()`** or the **`insert()`** member functions.

```
multimap<string, string> phonebook;
phonebook.emplace("Will", "555-1212");
phonebook.emplace("Will", "555-0123");
phonebook.emplace("Faye", "555-0707");
phonebook.emplace("Faye", "555-1234");
phonebook.emplace("Sarah", "555-8787");
phonebook.emplace("Sarah", "555-5678");
```

Getting the number of elements with the same key

- The `multimap` class's `count()` member function accepts a key as its argument, and returns the number of elements that match the specified key.

Program 17-20

```
1  #include <iostream>
2  #include <string>
3  #include <map>
4  using namespace std;
5
6  int main()
7  {
8     // Define a phonebook multimap.
9     multimap<string, string> phonebook =
10       { {"Will", "555-1212"},  {"Will", "555-0123"},
11         {"Faye", "555-0707"},  {"Faye", "555-1234"},
12         {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };
13
14    // Display the number of elements that match "Faye".
15    cout << "Faye has " << phonebook.count("Faye") << " elements.\n";
16    return 0;
17 }
```

Program Output

```
Faye has 2 elements.
```

Retrieving an element with specific key

- The multimap class has a find() member function that searches for an element with a specified key.
- The find() function returns an iterator to the first element matching it.
- If the element is not found, the find() function returns an iterator to the end of the multimap.
- To retrieve all elements matching a specified key, use the equal_range member function.
- The equal_range member function returns a pair object.
- The pair object's first member is an iterator pointing to the first element that matches the specified key.
- The pair object's second member is an iterator pointing to the position after the last element that matches the specified key.

Retrieving an element with specific key

```
// Define a phonebook multimap.
multimap<string, string> phonebook =
   { {"Will", "555-1212"},  {"Will", "555-0123"},
     {"Faye", "555-0707"},  {"Faye", "555-1234"},
     {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };

// Define a pair variable to receive the object that
// is returned from the equal_range member function.
pair<multimap<string, string>::iterator,
     multimap<string, string>::iterator> range;

// Define an iterator for the multimap.
multimap<string, string>::iterator iter;

// Get the range of elements that match "Faye".
range = phonebook.equal_range("Faye");

// Display all of the elements that match "Faye".
for (iter = range.first; iter != range.second; iter++)
{
   cout << iter->first << "\t" << iter->second << endl;
}
```

Program Output

```
Faye    555-0707
Faye    555-1234
```

Deleting an element with specific key

- To delete all elements matching a specified key, use the `erase()` member function.

```
// Define a phonebook multimap.
multimap<string, string> phonebook =
   { {"Will", "555-1212"},  {"Will", "555-0123"},
     {"Faye", "555-0707"},  {"Faye", "555-1234"},
     {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };

// Delete Will's phone numbers from the multimap.
phonebook.erase("Will");
```

unordered_multimap

- The `unordered_multimap` class is similar to the `multimap` class, except:
  - The keys in an `unordered_multimap` are not sorted
  - The `unordered_multimap` class has better performance
- You should use the `unordered_multimap` class instead of the `multimap` class if:
  - You will be making a lot of searches on a large number of elements
  - You are not concerned with retrieving them in key order
- The `unordered_multimap` class is declared in the `<unordered_multimap>` header file

The `set`, `multiset`, and `unordered_set` Classes

Sets

- A *set* is an associative container that is similar to a mathematical set.
- You can use the STL `set` class to create a set container.
- All the elements in a `set` must be unique. No two elements can have the same value.
- The elements in a set are automatically sorted in ascending order.
- The `set` class is declared in the `<set>` header file.

The set class

- You can use the STL `set` class to create a set container.
- The keys that are stored in a `map` container are unique – no duplicates.
- The `map` class is declared in the `<map>` header file.

Default Constructor	`set<dataType> name;` **Creates an empty `set`.**
Range Constructor	`set<dataType> name(iterator1, iterator2);` Creates a `set` that is initialized with a range of values. *`iterator1`* marks the beginning of the range and *`iterator2`* marks the end.
Copy Constructor	`set<dataType> name(set2);` Creates a `set` that is a copy of *`set2`*.

The set class

- Example: defining a `set` container to hold integers:

```
set<int> numbers;
```

- Example: defining and initializing a `set` container to hold integers:

```
set<int> numbers = {1, 2, 3, 4, 5};
```

- A set cannot contain duplicate items.
- If the same value appears more than once in an initialization list, it will be added to the set only one time.
- For example, the following set will contain the values 1, 2, 3, 4, and 5:

```
set<int> numbers = {1, 1, 2, 2, 3, 4, 5, 5, 5};
```

Set operations

- The insert() member function adds a new element to a set:

```
set<int> numbers;
numbers.insert(10);
numbers.insert(20);
numbers.insert(30);
```

```
// Create a set containing names.
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Display each element.
for (string element : names)
{
   cout << element << endl;
}
```

Iterators and sets

- The `begin()` and `end()` member functions return a bidirectional iterator of the `iterator` type
- The `cbegin()` and `cend()` member functions return a bidirectional iterator of the `const_iterator` type
- The `rbegin()` and `rend()` member functions return a reverse bidirectional iterator of the `reverse_iterator` type
- The `crbegin()` and `crend()` member functions return a reverse bidirectional iterator of the `const_reverse_iterator` type

```
// Create a set containing names.
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Create an iterator.
set<string>::iterator iter;

// Use the iterator to display each element in the set.
for (iter = names.begin(); iter != names.end(); iter++)
{
    cout << *iter << endl;
}
```

Determine whether and element exists

- The `set` class's `count()` member function accepts a value as its argument, and returns 1 if that value exists in the set. The function returns 0 otherwise.

```
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};
if (names.count("Lisa"))
   cout << "Lisa was found in the set.\n";
else
   cout << "Lisa was not found.\n";
```

Retrieving an element

- The `set` class has a `find()` member function that searches for an element with a specified value.
- The `find()` function returns an iterator to the element matching it.
- If the element is not found, the `find()` function returns an iterator to the end of the `set`.

```
// Create a set containing names.
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Create an iterator.
set<string>::iterator iter;

// Find "Karen".
iter = names.find("Karen");

// Display the result.
if (iter != names.end())
{
   cout << *iter << " was found.\n";
}
else
{
   cout << "Karen was not found.\n";
}
```

Storing objects of your own classes in a set

- If you want to store an object in a `set`, there is one requirement for that object's class:

  It must overload the < operator.

- Consider the following `Customer` class…

```
#ifndef CUSTOMER_H
#define CUSTOMER_H
#include<string>
using namespace std;

class Customer
{
private:
     int custNumber;
     string name;
public:
     Customer(int cn, string n)
     {  custNumber = cn;
        name = n; }

     void setCustNumber(int cn)
     {  custNumber = cn; }

     void setName(string n)
     {  name = n; }

     int getCustNumber() const
     {  return custNumber; }

   string getName() const
   {  return name; }

   bool operator < (const Customer &right) const
   {  bool status = false;

      if (custNumber < right.custNumber)
         status = true;

      return status; }
};
#endif
```

Storing objects of your own classes in a set

Program 17-22

```
#include <iostream>
#include <set>
#include "Customer.h"
using namespace std;

int main()
{
    // Create a set of Customer objects.
    set<Customer> customerset =
      { Customer(1003, "Megan Cruz"),
        Customer(1002, "Austin Hill"),
        Customer(1001, "Sarah Scott")
      };

    // Try to insert a duplicate customer number.
    customerset.emplace(1001, "Evan Smith");

    // Display the set elements
    cout << "List of customers:\n";
    for (auto element : customerset)
    {
       cout << element.getCustNumber() << " "
            << element.getName() << endl;
    }

    // Search for customer number 1002.
    cout << "\nSearching for Customer Number 1002:\n";
    auto it = customerset.find(Customer(1002, ""));

    if (it != customerset.end())
       cout << "Found: " << it->getName() << endl;
    else
       cout << "Not found.\n";

    return 0;
}
```

Program Output

```
List of customers:
1001 Sarah Scott
1002 Austin Hill
1003 Megan Cruz

Searching for Customer Number 1002:
Found: Austin Hill
```

multiset class

- The `mulitset` class is a set that allows duplicate items.
- The `mulitset` class has the same member functions as the `set` class (see Table 17-13 in your textbook).
- The `multiset` class is declared in the `<set>` header file.
- In the `set` class, the `count()` member function returns either 0 or 1. In the `multiset` class, the `count()` member function can return values greater than 1.
- In the `set` class, the `equal_range()` member function returns a range with, at most, one element. In the `multiset` class, the `equal_range()` member function can return a range with multiple elements.

Unordered_set class

- The unordered_set class is similar to the set class, except in two regards:
  - The values in an unordered_set are not sorted
  - The unordered_set class has better performance

- You should use the unordered_set class instead of the set class if:
  - You will be making a lot of searches on a large number of elements
  - You are not concerned with retrieving them in ascending order

- The unordered_set class is declared in the <unordered_set> header file

Algorithms

STL Algorithms

- The STL provides a number of algorithms, implemented as function templates, in the `<algorithm>` header file.
- These functions perform various operations on ranges of elements.
- A range of elements is a sequence of elements denoted by two iterators:
  - The first iterator points to the first element in the range
  - The second iterator points to the end of the range (the element to which the second iterator points is not included in the range).

Categories of Algorithms in the STL

- Min/max algorithms
- Sorting algorithms
- Search algorithms
- Read-only sequence algorithms
- Copying and moving algorithms
- Swapping algorithms
- Replacement algorithms
- Removal algorithms
- Reversal algorithms
- Fill algorithms
- Rotation algorithms
- Shuffling algorithms
- Set algorithms
- Transformation algorithm
- Partition algorithms
- Merge algorithms
- Permutation algorithms
- Heap algorithms
- Lexicographical comparison algorithm

Sorting

- The `sort` function:

  `sort(`*`iterator1`*`, `*`iterator2`*`);`

  `iterator1` and *`iterator2`* mark the beginning and end of a range of elements. The function sorts the range of elements in ascending order.

Searching

- The `binary_search` function:

  `binary_search(`*`iterator1, iterator2, value`*`);`

  `iterator1` and *`iterator2`* mark the beginning and end of a range of elements that are sorted in ascending order. value is the value to search for. The function returns `true` if value is found in the range, or `false` otherwise.

Program 17-23

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5
6  int main()
7  {
8     int searchValue;   // Value to search for
9
10    // Create a vector of unsorted integers.
11    vector<int> numbers = {10, 1, 9, 2, 8, 3, 7, 4, 6, 5};
12
13    // Sort the vector.
14    sort(numbers.begin(), numbers.end());
15
16    // Display the vector.
17    cout << "Here are the sorted values:\n";
18    for (auto element : numbers)
19       cout << element << " ";
20    cout << endl;
21
22    // Get the value to search for.
23    cout << "Enter a value to search for: ";
24    cin >> searchValue;
25
26    // Search for the value.
27    if (binary_search(numbers.begin(), numbers.end(), searchValue))
28       cout << "That value is in the vector.\n";
29    else
30       cout << "That value is not in the vector.\n";
31
32    return 0;
33 }
```

Program Output

```
Here are the sorted values:
1 2 3 4 5 6 7 8 9 10
Enter a value to search for: 8
That value is in the vector.
```

Program Output

```
Here are the sorted values:
1 2 3 4 5 6 7 8 9 10
Enter a value to search for: 99
That value is not in the vector.
```

Detecting Permutations

- If a range has *N* elements, there are *N*! possible arrangements, or permutations, of those elements.
- For example, the range of integers 1, 2, 3 has six possible permutations:

1, 2, 3
1, 3, 2
2, 1, 3
2, 3, 1
3, 1, 2
3, 2, 1

Detecting Permutations

- The `is_permutation()` function determines whether one range of elements is a permutation of another range of elements.

`is_permutation(`*`iterator1, iterator2, iterator3`*`)`

  - *`iterator1`* and *`iterator2`* mark the beginning and end of the first range of elements.
  - *`iterator3`* marks the beginning of the second range of elements, assumed to have the same number of elements as the first range.
  - The function returns `true` if the second range is a permutation of the first range, or `false` otherwise.
  - See Program 17-25 in your textbook for an example.

Plugging Your Own Functions into an Algorithm

- Many of the function templates in the STL are designed to accept function pointers as arguments.
- This allows you to "plug" one of your own functions into the algorithm.
- For example:

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

  - *iterator1* and *iterator2* mark the beginning and end of a range of elements.
  - *function* is the name of a function that accepts an element as its argument.
  - The `for_each()` function iterates over the range of elements, passing each element as an argument to *function*.

Plugging Your Own Functions into an Algorithm

```
void doubleNumber(int &n)
{
   n = n * 2;
}
```

```
vector<int> numbers = { 1, 2, 3, 4, 5 };

// Display the numbers before doubling.
for (auto element : numbers)
   cout << element << " ";
cout << endl;

// Double the value of each vector element.
for_each(numbers.begin(), numbers.end(), doubleNumber);

// Display the numbers before doubling.
for (auto element : numbers)
   cout << element << " ";
cout << endl;
```

Plugging Your Own Functions into an Algorithm

- Another example:

  `count_if(`*`iterator1, iterator2, function`*`)`

  - *`iterator1`* and *`iterator2`* mark the beginning and end of a range of elements.
  - *`function`* is the name of a function that accepts an element as its argument, and returns either `true` or `false`.
  - The `count_if()` function iterates over the range of elements, passing each element as an argument to *`function`*.
  - The `count_if` function returns the number of elements for which function returns `true`.

```
 1 #include <iostream>
 2 #include <vector>
 3 #include <algorithm>
 4 using namespace std;
 5
 6 // Function prototypes
 7 bool isNegative(int);
 8
 9 int main()
10 {
11    // Create a vector of ints.
12    vector<int> numbers = { 0, 99, 120, -33, 10, 8, -1, 101 };
13
14    // Get the number of elements that are negative.
15    int negatives = count_if(numbers.begin(), numbers.end(), isNegative);
16
17    // Display the results.
18    cout << "There are " << negatives << " negative elements.\n";
19    return 0;
20 }
21
22 // isNegative function
23 bool isNegative(int n)
24 {
25    bool status = false;
26
27    if (n < 0)
28       status = true;
29
30    return status;
31 }
```

Program Output

```
There are 2 negative elements.
```

Algorithms for Set Operations

- The STL provides function templates for basic mathematical set operations.

STL Function Template	Description
`set_union`	Finds the union of two sets, which is a set that contains all the elements of both sets, excluding duplicates.
`set_intersection`	Finds the intersection of two sets, which is a set that contains only the elements that are found in both sets.
`set_difference`	Finds the difference of two sets, which is the set of elements that appear in one set, but not the other.
`set_symmetric_difference`	Finds the symmetric difference of two sets, which is the set of elements that appear in one set, but not both.
`set_includes`	Determines whether one set includes another.

Introduction to Function Objects and Lambda Expressions

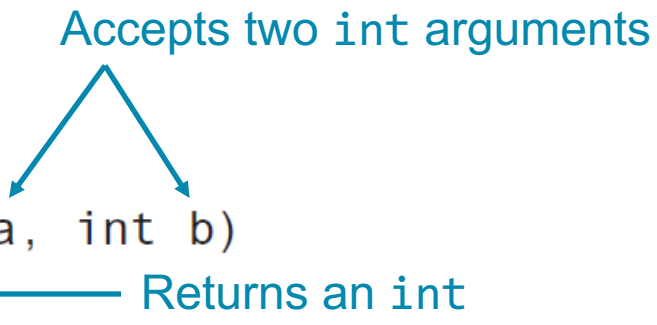
Function Objects

- A function object is an object that acts like a function.
  - It can be called
  - It can accept arguments
  - It can return a value
- Function objects are also known as *functors*

Function Objects

- To create a function object, you write a class that overloads the ( ) operator.

```
1   #ifndef SUM_H
2   #define SUM_H
3
4   class Sum
5   {
6   public:
7      int operator()(int a, int b)
8      { return a + b; }
9   };
10  #endif
```



Program 17-34

```
 1  #include <iostream>
 2  #include "Sum.h"
 3  using namespace std;
 4
 5  int main()
 6  {
 7     // Local variables
 8     int x = 10;
 9     int y = 2;
10     int z = 0;
11
12     // Create a Sum object.
13     Sum sum;
14
15     // Call the sum function object.
16     z = sum(x, y);
17
18     // Display the result.
19     cout << z << endl;
20
21     return 0;
22  }
```

Program Output

```
12
```

Anonymous Function Objects

- Function objects can be called at the point of their creation, without being given a name. Consider this class:

```
1   #ifndef IS_EVEN_H
2   #define IS_EVEN_H
3
4   class IsEven
5   {
6   public:
7      bool operator()(int x)
8      { return x % 2 == 0; }
9   };
10  #endif
```

Program 17-36

```
 1  #include <iostream>
 2  #include <vector>
 3  #include <algorithm>
 4  #include "IsEven.h"
 5  using namespace std;
 6
 7  int main()
 8  {
 9     // Create a vector of ints.
10     vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8 };
11
12     // Get the number of elements that even.
13     int evenNums = count_if(v.begin(), v.end(), IsEven());
14
15     // Display the results.
16     cout << "The vector contains " << evenNums << " even numbers.\n";
17     return 0;
18  }
```

An `IsEven` object is created here, but not given a name. It is anonymous.

Program Output

```
The vector contains 4 even numbers.
```

Predicate Terminology

- A function or function object that returns a Boolean value is called a *predicate*.
- A predicate that takes only one argument is called a *unary predicate*.
- A predicate that takes two arguments is called a *binary predicate*.
- This terminology is used in much of the available C++ documentation and literature.

Lambda Expressions

- A lambda expression is a compact way of creating a function object without having to write a class declaration.
- It is an expression that contains only the logic of the object's `operator()` member function.
- When the compiler encounters a lambda expression, it automatically generates a function object in memory, using the code that you provide in the lambda expression for the `operator()` member function.

Lambda Expressions

- General format:

  `[](`*`parameter list`*`) { `*`function body`*` }`

- The `[]` is known as the lambda introducer. It marks the beginning of a lambda expression.
- *`parameter list`* is a list of parameter declarations for the function object's `operator()` member function.
- *`function body`* is the code that should be the body of the object's `operator()` member function.

Lambda Expressions

- Example: a lambda expression for a function object that computes the sum of two integers:

```
[](int a, int b) { return a + b; }
```

- Example: a lambda expression for a function object that determines whether an integer is even is:

```
[](int x) { return x % 2 == 0; }
```

- Example: a lambda expression for a function object that takes an integer as input and prints the square of that integer:

```
[](int a) { cout << a * a << " "; }
```

Lambda Expressions

- When you call a lambda expression, you write a list of arguments, enclosed in parentheses, right after the expression.
- For example, the following code snippet displays 7, which is the sum of the variables x and y:

```
int x = 2;
int y = 5;
cout << [](int a, int b) {return a + b;}(x, y) << endl;
```

Lambda Expressions

- The following code segment counts the even numbers in a `vector`:

```
// Create a vector of ints.
vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8 };

// Get the number of elements that are even.
int evenNums = count_if(v.begin(), v.end(), [](int x) {return x % 2 == 0;});

// Display the results.
cout << "The vector contains " << evenNums << " even numbers.\n";
```

Lambda Expressions

- Because lambda expressions generate function objects, you can assign a lambda expression to a variable and then call it through the variable's name:

```
auto sum = [](int a, int b) {return a + b;};
int x = 2;
int y = 5;
int z = sum(x, y);
```

Program 17-37

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5
6  int main()
7  {
8     // Create a vector of ints.
9     vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8 };
10
11    // Use a lambda expression to create a function object.
12    auto isEven = [](int x) { return x % 2 == 0; };
13
14    // Get the number of elements that even.
15    int evenNums = count_if(v.begin(), v.end(), isEven);
16
17    // Display the results.
18    cout << "The vector contains " << evenNums << " even numbers.\n";
19    return 0;
20 }
```

Program Output

```
The vector contains 4 even numbers.
```

Functional Classes in the STL

- The STL library defines a number of classes that you can instantiate to create function objects in your program.
- To use these classes, you must `#include` the `<functional>` header file.
- Table 17-15 in your textbook lists a few of the functional classes:

Table 17-15 STL Function Object Classes

Functional Class	Description
`less<T>`	`less<T>()(T a, T b)` is true if and only if `a < b`
`less_equal<T>`	`less_equal()(T a, T b)` is true if an only if `a <= b`
`greater<T>`	`greater<T>()(T a, T b)` is true if and only if `a > b`
`greater_equal<T>`	`greater_equal<T>()(T a, T b)` is true if and only if `a >= b`