The Standard Template Library

Kafi Rahman

Assistant Professor @CS
Truman State University

17.6

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 a range 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.

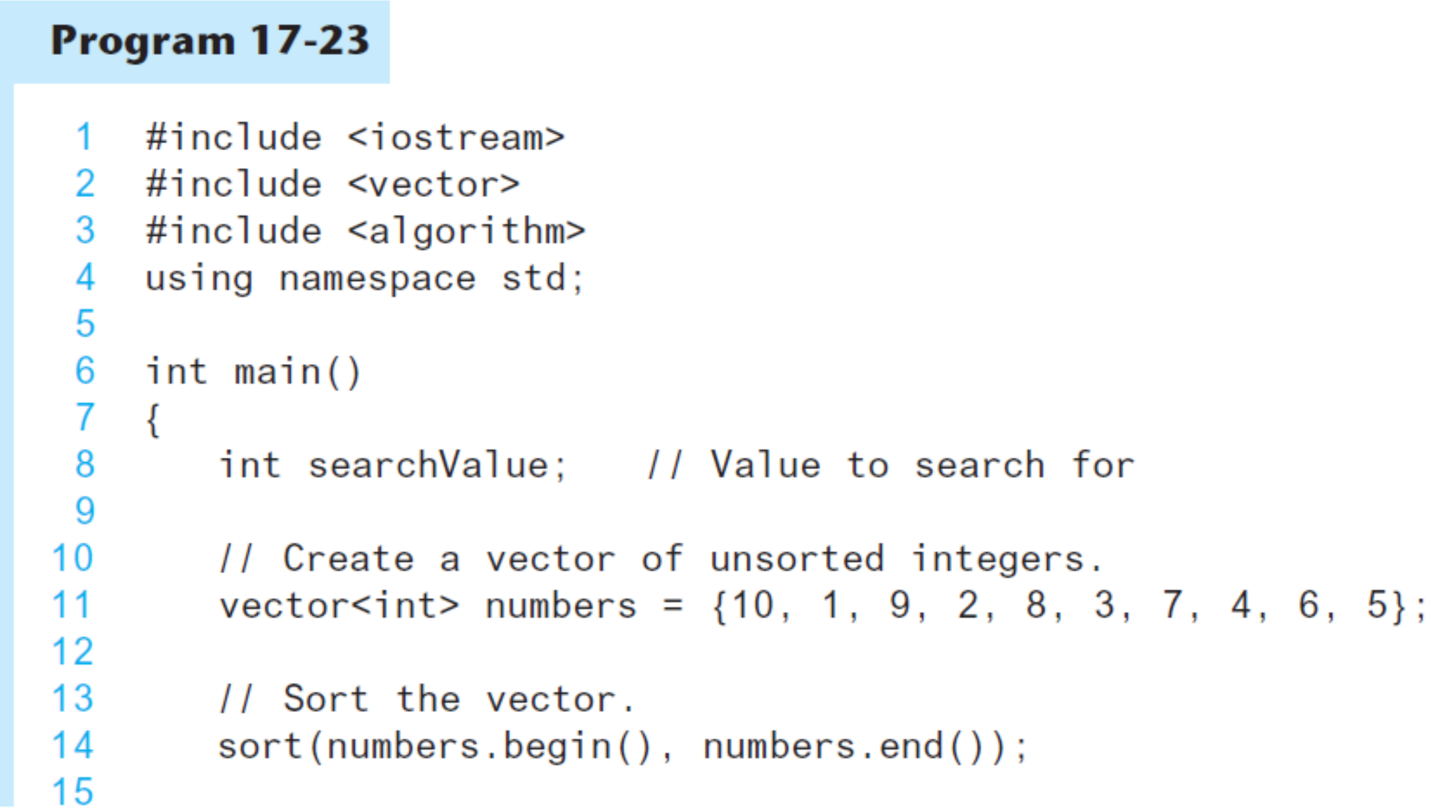
Searching

- 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.

Searching (cont)

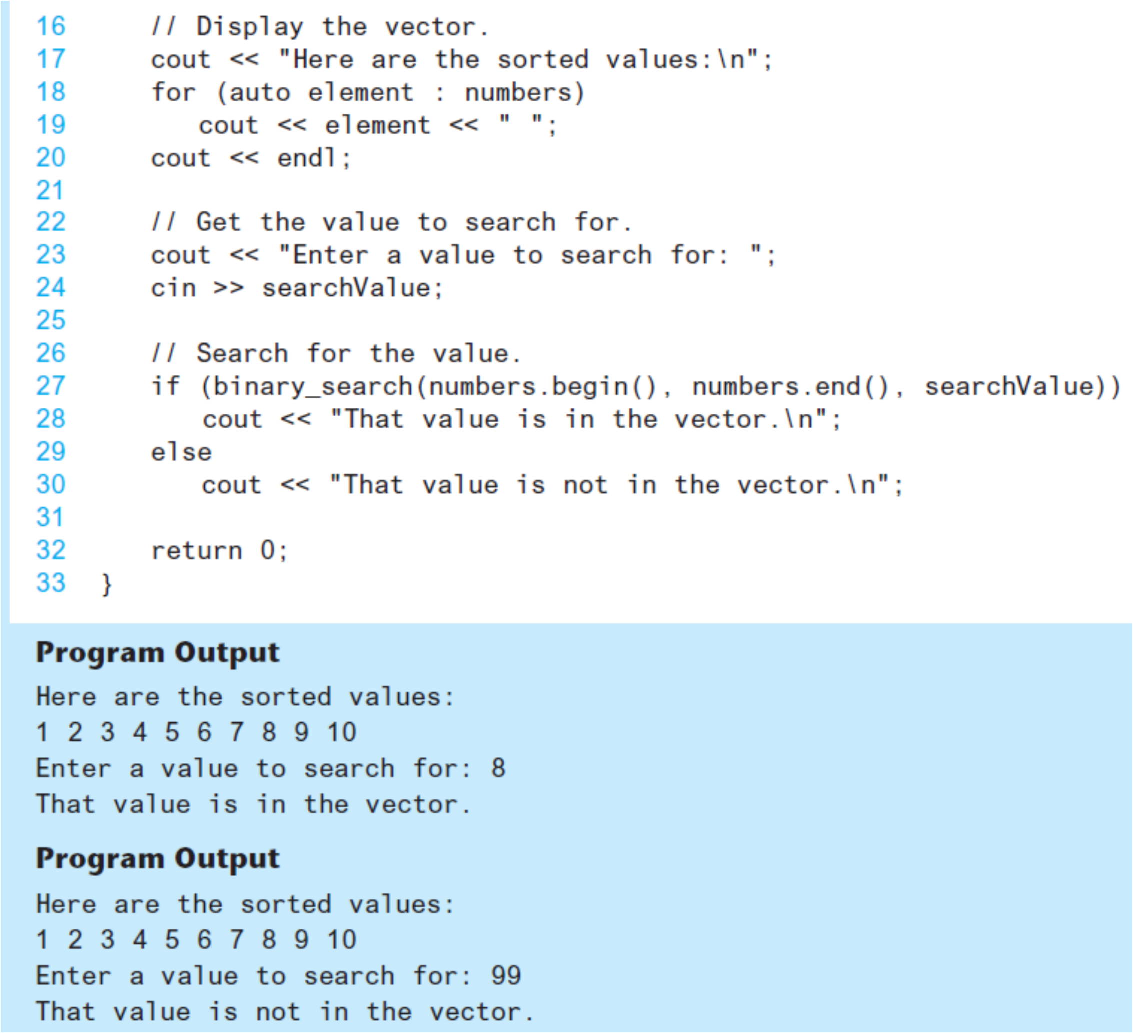


Program 17-23

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

int main()
{
    int searchValue;    // Value to search for

    // Create a vector of unsorted integers.
    vector<int> numbers = {10, 1, 9, 2, 8, 3, 7, 4, 6, 5};

    // Sort the vector.
    sort(numbers.begin(), numbers.end());

```

Continued…

Searching (cont)



```
    // Display the vector.
    cout << "Here are the sorted values:\n";
    for (auto element : numbers)
        cout << element << " ";
    cout << endl;

    // Get the value to search for.
    cout << "Enter a value to search for: ";
    cin >> searchValue;

    // Search for the value.
    if (binary_search(numbers.begin(), numbers.end(), searchValue))
        cout << "That value is in the vector.\n";
    else
        cout << "That value is not in the vector.\n";

    return 0;
}
```

Program Output

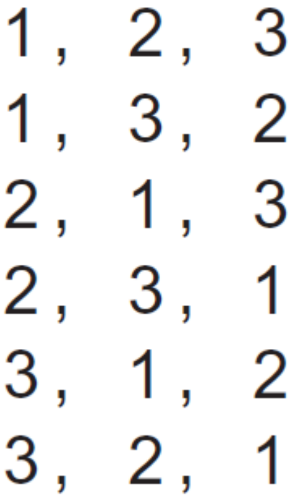
```
Here are the sorted values:
1 2 3 4 5 6 7 8 9 10
Enter a value to search for: 8
That value is in the vector.
```

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, 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.

Detecting Permutations: program code

```
std::vector<int> series = {1, 2, 3};
std::vector<int> testSeries = {3, 1, 2};

if (is_permutation(series.begin(), series.end(),
testSeries.begin()))
{
    cout << "\nYes, they are permutation";
}
else
{
    cout << "\nThey are not permutation";
}
```

Plugging Your Own Functions into an Algorithm

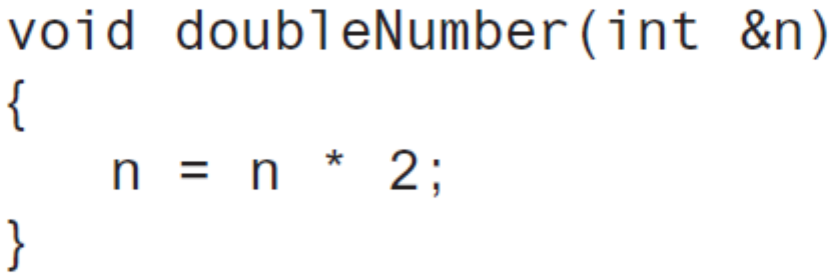
- Many of the function templates in the STL are designed to accept function pointers as arguments.
- This allows you to "plug" one of your own functions into the algorithm.
- For example:

  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.

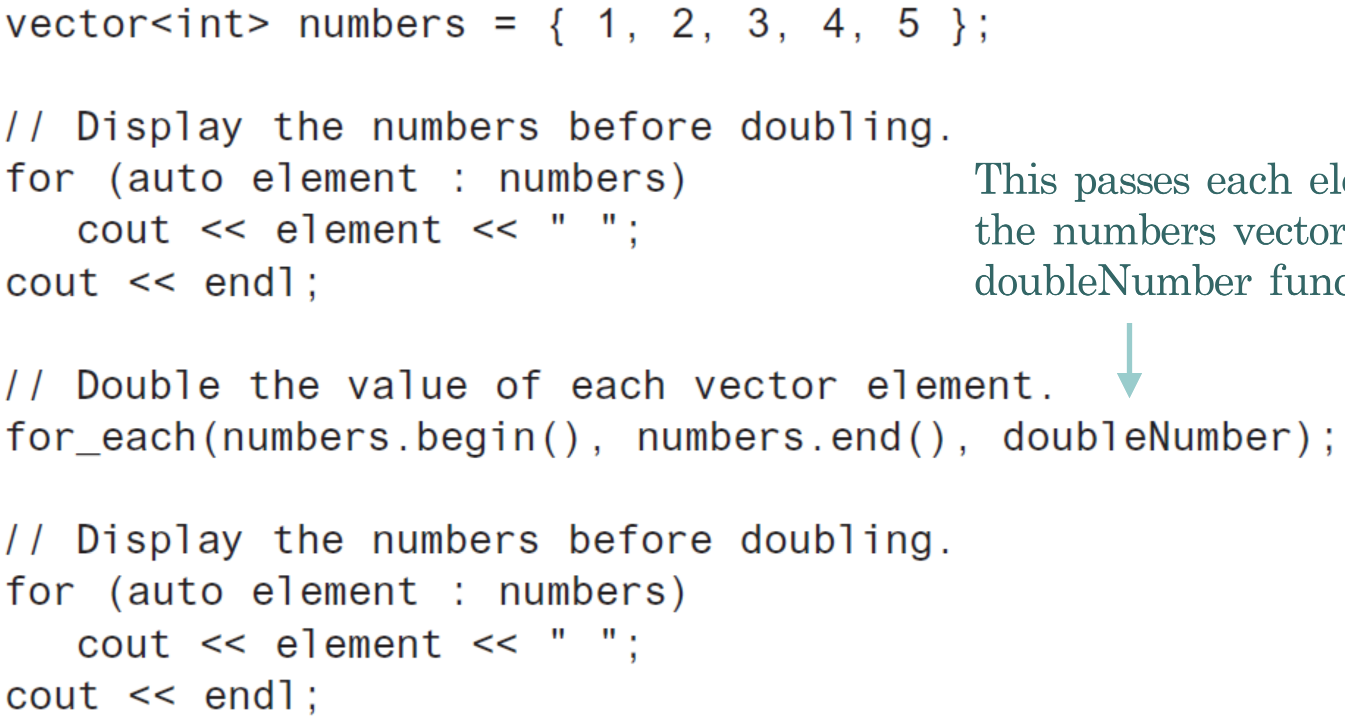
Plugging Your Own Functions into an Algorithm

- For example, consider this function:



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

Plugging Your Own Functions into an Algorithm



```
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;
```

This passes each element of the numbers vector to the doubleNumber function.

- the function will take only one parameter
- the parameter type must match

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.

Plugging Your Own Functions into an Algorithm

```cpp
// Function prototypes
bool isNegative(int);

int main()
{
     // Create a vector of ints.
     vector<int> numbers = { 0, 99, 120, -33,
10, 8, -1, 101 };

     // Get the number of elements that are
negative.
     int negatives = count_if(numbers.begin(),
numbers.end(), isNegative);
     // Display the results.
     cout << "There are " << negatives << "
negative elements.\n";
     return 0;
}
```

```cpp
// isNegative function
bool isNegative(int n)
{
     bool status = false;

     if (n < 0)
     status = true;

     return status;
}
```

Program Output: There are 2 negative elements.

Algorithms for Set Operations

```
vector<int> foo{100, 200, 300};
vector<int> bar{1, 2, 3, 4, 5};

// when inserting in a sequence container, insertion
point advances
// because each std::insert_iterator::operator= updates
the target iterator
copy(foo.begin(), foo.end(), inserter(bar, bar.begin()));

//bar.insert(bar.begin(), foo.begin(), foo.end());

for (int n : bar) // display all the values from bar
    cout << n << ' ';
```

- inserter function uses the range of values from foo.begin() to foo.end() and adds them in the bar variable (starting from the begin() position)

Algorithms for 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.

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

Algorithms for Set Operations: set_union

```
set<int> oneSet = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
set<int> twoSet = {5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};

set<int> resultSet;

// set_union: all the elements from both sets (no duplicates)
set_union(oneSet.begin(), oneSet.end(),            //first set
          twoSet.begin(), twoSet.end(),            //second set
          inserter(resultSet, resultSet.begin())); // result set

for (auto val : resultSet) // displaying all the values
    cout << val << " ";
```

- Output: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Algorithms for Set Operations: set_intersection

```
set<int> oneSet = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
set<int> twoSet = {5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};

set<int> resultSet;

// set_intersection: elements that are in both sets
set_intersection(oneSet.begin(), oneSet.end(),        //first set
          twoSet.begin(), twoSet.end(),                //second set
          inserter(resultSet, resultSet.begin()));     //result set

for (auto val : resultSet) // displaying all the values
    cout << val << " ";
```

- Output: 5 6 7 8 9 10

Algorithms for Set Operations: set_symmetric_difference

```
set<int> oneSet = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
set<int> twoSet = {5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};

set<int> resultSet;

// set_symmetric_difference: appear in either set but not both
set_symmetric_difference(oneSet.begin(), oneSet.end(), //first set
          twoSet.begin(), twoSet.end(),                //second set
          inserter(resultSet, resultSet.begin()));     //result set

for (auto val : resultSet) // displaying all the values
    cout << val << " ";
```

- Output: 1 2 3 4 11 12 13 14 15

Algorithms for Set Operations: set_difference

```
set<int> oneSet = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
set<int> twoSet = {5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};

set<int> resultSet;

// set_difference: appear in first set but not the second
set_difference(oneSet.begin(), oneSet.end(),          //first set
          twoSet.begin(), twoSet.end(),               //second set
          inserter(resultSet, resultSet.begin()));    //result set

for (auto val : resultSet) // displaying all the values
    cout << val << " ";
```

- Output: 1 2 3 4

Wishing you and your family

Happy Easter