

LABORATORY 12

Vectoring in on vectors

Objective

C++ imposes significant restrictions on the use of arrays—a function return type cannot be an array; an array cannot be passed by value; an array cannot be the target of an assignment; the size of the array must be a compile-time constant; and an array cannot be resized. The restrictions on arrays forced the developers of many software applications to use alternative list representations that were often nonportable. The cost of using nonportable representations could be quite high because developers had to create and support multiple versions of their software. This expense can now be avoided by using the container classes of the Standard Template Library (STL). For many programming situations, the appropriate container class to use is the `vector` class. Therefore, we explore the `vector` class in lab this week.

- Vectors
- Subscript and `at()`
- Resizing
- Passing vectors
- Iterators
- Sorting

12.1 GETTING STARTED

- Using the procedures in the introductory laboratory handout, create the working directory `\cpplab` on the appropriate disk drive and obtain a copy of self-extracting archive `lab12.exe`. The copy should be placed in the `cpplab` directory. Execute the copy to extract the files necessary for this laboratory.
- Many of the activities that are performed in the laboratory can be done in groups but you should work the exercises yourself.

12.2 VECTOR BASICS

The container classes of the STL are a set of generic list representations that allow programmers to specify which types of elements their particular lists are to hold. The `vector` class template is the most widely used representation. The principal member functions of the `vector` class are given in Table 12.1.

We start off by having you manipulate a vector using the indexing members—the subscript operator `[]` and the `at()` function.

- Open the program file `basics.cpp`. The program in this file defines, initializes, and displays a vector `A`. (For your information, the elements of the vector are set to the first fifteen values in the Fibonacci sequence). Create a default workspace and execute the program to see a sample run.
- Add three assignment statements to the program after the setup of `A`. The assignment statements should use the subscript operator to make the first, second, and fourth elements of `A` equal to 100. (Remember the first element has index 0). The subscript operator when applied to a vector returns a reference, thus making the result assignable. Run the program to verify that your assignments are correct.
- Modify the display of `A` so rather than using the vector subscript operator to access an element, the code uses the vector `at()` member function. Run the program to verify that your modifications are correct.
- Like the subscript operator, the `at()` member function returns a reference rather than a simple value. Therefore, its result can be the target of an assignment as in the following statement.

```
A.at(9) = 12;
```



- Add three assignment statements to the program that set the last three elements of `A` to 200. The assignments should occur immediately before the loop that displays the elements. ✓
- You may wonder why the vector class provides two mechanisms for indexing the elements of a list. The `at()` function is supposed to check that the index is proper (i.e., it is in the interval $0 \dots \text{size}() - 1$). If the index is valid, `at()` returns a reference to the desired list element; otherwise `at()` is supposed to throw an exception. Some compilers have not implemented `at()` as it is defined in the standard. Test whether your compiler follows the standard for member function `at()`. Add the following assignment statement to the program.

```
A.at(-1) = 1954;
```



- Run your modified program. Does your program throw an exception for the invalid subscript? ✓

- Close the current workspace.

Table 12.1

Some member functions of the class template vector

<code>vector::vector()</code>	The default constructor creates a vector of 0 length.
<code>vector::vector(const T &V)</code>	The copy constructor creates a vector that is a duplicate of vector <i>V</i> .
<code>vector::vector(size_type n, const T &val = T())</code>	Explicit constructor creates a vector of length <i>n</i> with each element initialized to <i>val</i> .
<code>size_type size() const</code>	Returns the numbers of elements in the vector.
<code>iterator insert(iterator pos, const T &val = T())</code>	Inserts a copy of <i>val</i> at position <i>pos</i> of the vector and returns the position of the copy into the vector.
<code>iterator erase(iterator pos)</code>	Removes the element of the vector at position <i>pos</i> .
<code>void pop_back()</code>	Removes the last element of the vector.
<code>void push_back(const T &val)</code>	Inserts a copy of <i>val</i> after the last element of the vector.
<code>void resize(size_type s, T val = T())</code>	Let <i>n</i> be the current number of elements in the vector. If <i>s</i> > <i>n</i> , then the number of elements is increased to <i>s</i> with the new elements added after the existing elements and the initial value of the new elements being <i>val</i> . If <i>s</i> < <i>n</i> , then the number of elements is decreased to <i>s</i> by erasing elements from the end of the vector. If <i>s</i> equals <i>n</i> , then no action is taken.
<code>void vector::clear()</code>	Removes all elements from the vector.
<code>reference at(int i)</code>	If <i>i</i> is a valid index, it returns the <i>i</i> th element; otherwise an exception is thrown.
<code>const_reference at(int i)</code>	If <i>i</i> is a valid index, it returns the <i>i</i> th element; otherwise an exception is thrown. The element that is returned cannot be modified.
<code>iterator begin()</code>	Returns an iterator pointing to the first element of the vector.
<code>const_iterator begin()</code>	Returns an iterator pointing to the first element of the vector. Elements dereferenced by this iterator cannot be modified.
<code>iterator end()</code>	Returns an iterator pointing to a sentinel immediately beyond the last element.
<code>const_iterator end()</code>	Returns an iterator pointing to a sentinel immediately beyond the last element. Elements dereferenced by this iterator cannot be modified.

- Open the file `numbers.cpp`. The program in this file extracts values from the standard input stream to set the elements of a vector. This short program demonstrates in part the ability of the vector container class to resize itself.
- Create a default workspace and run the program on the following data set.
6 21 54 6 30 54
- Now run the program on the following data set.
6 9 82 11 28 85 11 29 91

Unlike the program contained in `five.cpp` of the previous lab, we do not need to modify constants and recompile `numbers.cpp` to correctly handle a different-sized list. The vector class, with its use of dynamic data structures, can handle different-sized lists automatically.

- Modify the program so that immediately after a new value is added to the list, the program displays the current size of the list.
- Modify the program so that two duplicates of A are made. The duplicates should be named B and C. Duplicate B should be built at the same time as A is being built. Duplicate C should be built after all of the inputs have been extracted. Duplicate C should be built without a loop.



- Modify the program so duplicates B and C are also displayed. ✓
- Close the current workspace.

12.3 SOME SIMPLE FUNCTIONS

- Open the file `functions.cpp` and create a default workspace. This file defines a vector of strings S and a vector of integers N.
- Your first task is to write functions, `DisplayStrings()` and `DisplayInts()`, that display to a desired stream respectively lists of type `vector<string>` and `vector<int>`. The functions take two parameters. For both functions, the first parameter is a reference to an `ostream`. For one function, the second parameter is a constant reference parameter of type `vector<string>`; for the other function, the second parameter is a constant reference parameter of type `vector<int>`.

The prototypes of the functions are:

```
void DisplayStrings(ostream &sout,
    const vector<string> &A);
void DisplayInts(ostream &sout,
    const vector<int> &A);
```

The functions should perform the following actions.

- Step 1.* Display the left bracket and whitespace
- Step 2.* For each vector element do
 - Step 2.1* Display the element in an appropriate manner

Step 2.2 Display whitespace


Step 3. Display the right bracket

A sample run of the program follows.



```

MS-DOS Prompt
8 x 12
C:\cpp\lab> functions
S: [ "goo goo" "cinders" "lady" "nilla" "galen" ]
N: [ 1 4 32 5 12 28 6 21 54 8 21 76 9 28 29 30 ]

```


- 
 - Show the laboratory instructor your implementation of the two vector display functions and a sample run. ✓
 - Using functions to display objects is not the norm in C++. Instead programmers normally use the insertion operator. With your display functions as a basis, implement overloading of the insertion operator for `vector<string>` and `vector<int>` objects. (Remember these insertion operators need to perform a reference return of their stream parameter). The prototypes for these operators are:


```

ostream& operator<<(ostream &sout,
    const vector<string> &A);
ostream& operator<<(ostream &sout,
    const vector<int> &A);
          
```
- 
 - Uncomment the vector insertions and show the laboratory instructor your implementation of the two insertion operators and a sample run. ✓
 - The *mean* of a list of values is their average. Develop a function `mean()` that first asserts that its constant `vector<int>` reference parameter `A` has at least one element. The function next sums the values in `A`. The function then returns that sum divided by the number of elements in `A`. Add your definition to the program file.
- 
 - Uncomment the invocation of `mean()` in function `main()`. Show your results to the laboratory instructor. ✓

The vector member function `resize()` gives you the ability to add or remove elements from the end of the list. Function `resize()` has a required parameter that indicates the new size of the list. The function also has an optional parameter that specifies the initial value of any new elements that are created by resizing the list. If the optional parameter is not specified, the new elements are initialized using the default constructor for the base type of the vector. (For fundamental base types, 0 is used).

- Immediately after the definitions and assignments to `S` and `N` in `functions.cpp`, perform separate `resize()` invocations to implement the following list manipulations. After each manipulation, display the modified list using your insertion operators.

- Resize list S to 9 elements. The new elements should have the value "Darby".
- Resize list N to 20 elements. Do not provide a second parameter.
- Resize list N to 10 elements.
- Resize list N to 15 elements using the value 8 as the initial value of new elements.
- Resize list S to 0 elements. Use "buffer" as the value of the second parameter.
-  ▪ Show the laboratory instructor your resizing code and a sample run. ✓
- Close the current workspace.

12.4 ITERATORS

The STL library provides an alternative method to reference the elements of a vector. The alternative method uses iterators, where an iterator is conceptually a pointer to an element in the list. There are three basic operations on iterators.

- increment operator ++: updates the iterator to point to the next element in the list. If there is no next element, the iterator points to a sentinel.
- decrement operator --: updates the iterator to point to the previous element in the list.
- dereferencing operator *: produces a reference to the element to which the iterator points.

The vector member functions `begin()` and `end()` described in Table 12.1 return iterators when they are invoked. There is an iterator type for each type of vector. For example, the iterator type associated with a `vector<int>` container is `vector<int>::iterator` and the iterator type associated with a `vector<string>` container is `vector<string>::iterator`.

- Open the file `iterator.cpp` and create a default workspace.
- Because the type names `vector<int>::iterator` and `vector<string>::iterator` are unwieldy, programmers often add `typedef` statements to their program that allow simpler type names to be used. For example, the following statements allow us to use `string_iterator` for `vector<string>::iterator` and `int_iterator` for `vector<int>::iterator`.

```
typedef vector<int>::iterator int_iterator;
typedef vector<string>::iterator string_iterator;
```

Add `typedef` statements prior to function `main()` to create simpler iterator type names.

- Define four iterators in the program file. The iterators should be defined after A and B have been displayed.
 - An iterator P that points to the first element of A.

- An iterator Q that points to the trailing sentinel for A.
- An iterator R that points to the first element of B.
- An iterator S that points to the trailing sentinel for B.
- Modify the element that iterator P points to by using the dereferencing operator *. The element's new value is 29.
- Modify the element that iterator R points to by using the dereferencing operator *. The element's new value is "merlin".
- After these modifications, add statements that cause iterators P and R to be incremented and iterators Q and S to be decremented.
- After the previous modifications, add statements to the program file that modify the element to which iterator P points. The element's new value is 31. Similarly, modify the element to which iterator Q points so that it has the value 85. Also modify the elements to which iterators R and S point. The new element values are respectively "snooky" and "hennepin".
- If necessary, add insertion statements that display lists A and B after these changes have been made. Are the values the ones that you expect? If not, examine your code and the definitions of the operators.
- The most common use of iterators is in loops. For example, the following code segment displays the elements of A one per line.

```
string_iterator s = B.begin();
while (s != B.end()) {
    cout << *s << endl;
    ++s;
}
```

Add a code segment using iterators that determines the string that occurs first lexicographically (i.e., determine the string that has the minimum string value). The code segment should display the value of that string after processing all of the loop elements.



- Show your program file along with a run of the program to your laboratory instructor. ✓
- Close the current workspace.

12.5 SORTING

One of the better sorting methods is `MergeSort()`. This recursive sort divides a list of n elements into two sublists of size $n/2$. The sublists are sorted by recursive calls to `MergeSort()`. After the two sublists are sorted, they are merged together to produce a single sorted list of size n . The function `Merge-`

Sort() from mergesort.cpp is an implementation of this brief description.

```
void MergeSort(vector<int> &A, int left, int right) {
    int size = right - left + 1;
    int mid = (left + right)/2;
    if (size > 2) {
        MergeSort(A, left, mid);
        MergeSort(A, mid + 1, right);
    }
    if (size > 1) {
        Merge(A, left, mid, right);
    }
}
```

Thus it is repeated calls to function Merge() by MergeSort() that actually put the elements into sorted order. Function Merge() takes four parameters. The first parameter is the vector A that contains the two sorted sublists to be merged. The next three parameters are indices left, mid, and right. The left sorted sublist consists of the elements A[left] ... A[mid]; the right sorted sublist consists of the elements A[mid+1] ... A[right].

- Open mergesort.cpp and create a default workspace. Finish the implementation of this sorting method by completing the implementation of function Merge(). For code simplicity, our implementation of the function first puts the merging of the A sublists into a list B. List B is then copied to the appropriate elements of list A. Our implementation is based on the following algorithm.

Step 1. Create temporary vector B for merging the sorted sublists.

Step 2. Set up and initialize indices into the A sublists and the list B.

Step 3. Repeatedly perform the following while both sublists of A have uncopied elements to consider.

Step 3.1 If the current element in the right sublist is smaller than the current element in the left sublist, copy the element from the right sublist to the next available position in B. Update the indices of the right sublist and the B list to index the successive elements.

Step 3.2 If instead the current element in the left sublist is smaller than the current element in the right sublist, copy the element from the left sublist to the next available position in B. Update the indices of the left sublist and the B list to index the successive elements.

Step 4. If the left sublist has uncopied elements, copy them to B.

Step 5. If instead the right sublist has uncopied elements, copy them to B.

Step 6. Copy B to A.



- Show the laboratory instructor your program and a sample run. ✓

In the previous lab we measured the quality of a searching method by determining the number of element comparisons it made. In evaluating the quality of a sorting method, it is standard to use the number of element comparisons *and* the number of element assignments made by the method. Generally these

two actions are the most time-consuming operations a sorting method performs.

- Define two global integer objects `comps` and `moves` at the beginning of `mergesort.cpp`. Initialize both objects to 0. Object `comps` will keep track of the number of element comparisons made during the sort; object `moves` will keep track of the number of element assignments made during the sort.
- To have `comps` and `moves` appropriately maintained by the program, you must make some additions to function `Merge()`. There is one element comparison and one element move made per iteration of the first while loop. Therefore, add the following statements to the beginning of the while loop body.

```
++comps;
++moves;
```

- The remaining `Merge()` code does not make any element comparisons. However, the other three loops perform an element assignment for each of their iterations. Therefore, add an increment of `moves` to the bodies of those loops.
- Add statements to display the final values of `comps` and `moves`.
- Run your program four times. The first time have the program sort the first 2 elements of `N`. The second time have the program sort the first 4 elements of `N`. The third time have the program sort the first 8 elements of `N`. The last time have the program sort all 16 elements of `N`. To perform these different actions, you will need to modify the invocation of `MergeSort()` in function `main()`. Record the number of comparisons and element assignments in the following table.

elements	comps	moves
2		
4		
8		
16		



- Show your laboratory instructor your results. Speculate on the pattern that is developing for the values of `comps` and `moves`. ✓

12.6 FINISHING UP

- Copy any files you wish to keep to your own drive.
- Delete the directory `\cpplab`.
- Hand in your check-off sheet.