Lab 7.1 Templates—Introduction to `vectors`

Project 7.1 `vectors` of System Tasks

Project 7.2 Building and Using a Queue Template

NOTES TO THE INSTRUCTOR

The purpose of this lab exercise is to take a close look at the `vector` container from the Standard Template Library. It may have already been introduced in a first programming course in place of or in addition to the more traditional study of arrays. This lab exercise, however, will take a more detailed look at this container:

- As a standard example of a class template
- To illustrate how an "old-fashioned" container type like the array can be made into an object-oriented structure, a generic self-contained class template
- How it is used—the operations it provides, including the more advanced and specialized ones not typically covered in a first programming course
- A look "under the hood" at how it grows when necessary so that it can store more values

The lab exercise parallels closely the presentation in Section 9.4 of the text *ADTs, Data Structures, and Problem Solving with C++*, 2E.

Two different projects are provided:

Project 7.1 shows the truly generic nature of a class template such as `vector` by considering `vectors` of various kinds of system tasks. It also demonstrates the generic nature of STL's algorithms by showing how the `sort` algorithm can be used to sort all of these different types of vectors, which in turn demonstrates the importance of being able to overload operators on classes—`operator<()` in particular for the `sort()` algorithm—and introduces iterators, which make it possible for STL's algorithms to operate on a variety of different containers.

Project 7.2 is intended to show what is involved in converting a class into a class template and its generic nature. In *Part I: Building a Queue Template*, the `Queue` class developed in Project 6.1 (or Project 6.2) is to be converted into a class template that has the same operations as before plus an output operator (<<):

- constructor
- empty
- enqueue
- front
- dequeue
- destructor
- copy constructor
- assignment operator
- output operator (<<)

A driver program to test the class template must also be written.

Part II: Application: Queues From All Over! Enough to Drive You Crazy!!! is similar to Project 7.1 but it uses the `Queue` class template developed in Part I rather than `vector`. The generic nature of class templates is demonstrated by considering queues of various kinds—queues of `doubles`; queues of strings; queues whose elements are `Student` objects, where `Student` is a small class built by the students; queues whose elements are pointers to `Students`; and queues of queues.

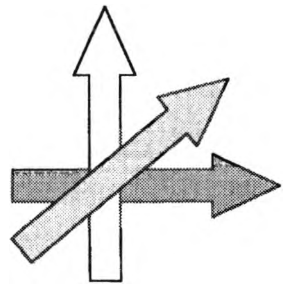
Notes:

1. Lab Exercise 7.1 uses the program `vectorlab.cpp` to investigate various features of the `vector` container. This file can be downloaded from the website whose URL is given in the preface.

2. It is not intended that both projects will be assigned. You may select one (or a part of one) that best fits your preferences and course content.

Course Info: ________________ ***Name:*** Chris Dang

Lab 7.1 Templates—Introduction to `vectors`



Background: In Lab Exercise 3.1 we saw that one of the problems of an array is that it doesn't know how large it is; also, most compilers allow indices to exceed the memory space allocated for the array. Arrays simply don't know how large they are. One of the principles of object oriented programming is that objects need to be self-contained. One shouldn't have to go outside the object itself to discover information about it. The `vector` class template eliminates this difficulty. You should study Section 9.4 of the textbook *ADTs, Data Structures, and Problem Solving with C++*, 2E before or while working through this lab exercise.

The Template Concept

The template idea is important because it allows code to be written that is generic in data type. Thus, for example, we can develop a data structure that doesn't care what kind of data is in the structure until it is declared. Not only is this mechanism used in a large and useful library of standard templates called *STL* (the *Standard Template Library*) but we can also use it to develop our own templates.

Template code is written with a generic place holder for data types that will be replaced with a specific type when the template is used. Thus if you needed a sort function, you could write it as a template and then sort data types of any kind for which the less-than (`<`) operator is defined just by specifying the type to be used in a function call. Section 9.2 of the text describes such *function templates*. In this lab we consider one example of a *class template*.

The `vector` *Class Template*

The STL contains three kinds of components: 1) *containers*, 2) *iterators*, and 3) *algorithms*—all of which are generic components. You will use the `vector` container in this lab to become familiar with templates and their uses.

A `vector` is an object-oriented counterpart of a one-dimensional array. It has several advantages over a standard C-type array:

- Like an array, it can store and process any type of data.
- Unlike a standard array, it knows how large it is.
- It can dynamically increase its capacity.

As a class template, its declaration has the (simplified) form:

```
template <typename T>
class vector
{
  // details of vector
};
```

where `T` is a parameter for the type of values to be stored in the container. So when we actually declare a `vector` in a program we have to say what actual type `T` represents. For example:

`vector<double> vec1;`	Creates a vector `vec1` of doubles, and the compiler generates the correct function members
`vector<string> name;`	Creates a vector `name` of strings, and the compiler generates the correct function members

You will be using the program `vectorlab.cpp` to experiment with `vectors`. You should download this file using the procedure described by your instructor. (It can be downloaded from the website whose URL is given in the preface.)

Getting Started with `vectors`

Before we can use a `vector`, we have to declare one. The `vector` class template provides four constructors and a copy constructor, so there are several ways to do this:

```
vector<element_type> object_name;        // default constructor

vector<element_type> object_name(initial_capacity);
                                                  //explicit-value #1

vector<element_type> object_name(initial_capacity, initial_value);
                                                  //explicit-value #2

vector<element_type> object_name(first_ptr, last_ptr);
```

where *`first_ptr`* and *`last_ptr`* are pointers to (i.e., the addresses of) array elements

Some Example Definitions from `vectorlab.cpp`

```
vector<int> v1;

vector<int> v2(2);

int numInts;
cin >> numInts;
vector<int> v3(numInts);   // run-time specification of capacity

int a[]={1, 4, 9, 16, 25};
vector<int> v5(a, a + 5); // constructs vector v5 to contain the array
                          // elements a[0], a[1], ..., a[4]
```

Some Member Functions of `vector` *Dealing with Capacity and Size*

`v.empty()`	Returns `true` if and only if v contains no values
`v.size()`	Returns the number of values v contains
`v.capacity()`	Returns the capacity of v, that is, the number of values it can currently store
`v.max_size()`	Returns the maximum number of elements v can have (i.e., the maximum capacity)
`v.reserve(n)`	*Increases* the capacity of v to n (does not affect `size()`)—*cannot be used to decrease capacity*

Using `vectorlab.cpp` *as a Workbench to Exercise* `vectors`

Now you will make additions to `vectorlab.cpp` in order to answer a series of questions and learn how `vectors` behave.

Note: You will be outputting a lot of vectors and information about them. Be sure to label what each output is!

[1] In the section of `vectorlab.cpp` labeled `//--- 1 ---` you are to add statements that display the capacity and size of each of `v1`, `v2`, `v3`, `v4`, and `v5` and whether each is empty.

Write the necessary statement(s) for v1 below.

Now add this and similar statements for v2, v3, v4, and v5 to vectorlab.cpp in the section labeled //--- 1 ---. Then compile and execute vectorlab.cpp and record the results in the following table:

vector	capacity	size	empty (Y|N)
v1	0	0	Y
v2	2	2	N
v3	10	10	N
v4	3	3	N
v5	5	5	N

[2] The maximum capacity of a vector is machine dependent. In the section of vectorlab.cpp labeled //--- 2 ---, add a statement to determine the maximum capacity of v1 for your system and record this value here: 1073741823

[3] In the section labeled //--- 3 ---, add the statement v4.reserve(7); and an output statement to display its capacity and size. Record them here:

vector	capacity	size
v4	7	3

[4] The output operator has been overloaded in vectorlab.cpp so it can be used to output vector<T>s.

In the section labeled //--- 4 ---, add output statements to display each of v1, v2, v3, v4, and v5 and record the contents of each below:

v1: cout << v1 << endl ;

v2: cout << v2 << endl ;

v3: cout << v3 << endl ;

v4: cout << v4 << endl ;

v5: cout << v5 << endl ;

Let's recap what has happened:

The first declaration

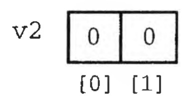
```
vector<int> v1;
```

constructs `v1` as an empty `vector<int>`.

The second and third declarations

```
vector<int> v2(2);
int numInts;
cout << "Enter capacity of v3: ";
cin >> numInts;
vector<int> v3(numInts);
```

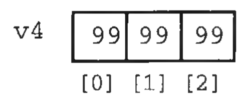
construct `vector<int>`s with specified capacities and with size = capacity and all elements initialized to the default value `0`; for example:



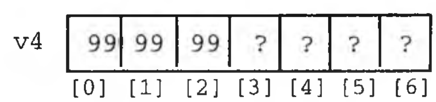
The fourth declaration

```
vector<int> v4(3, 99);
```

constructs `v4` as a `vector<int>` with capacity 3 and size 3 and initializes each element to `99`:



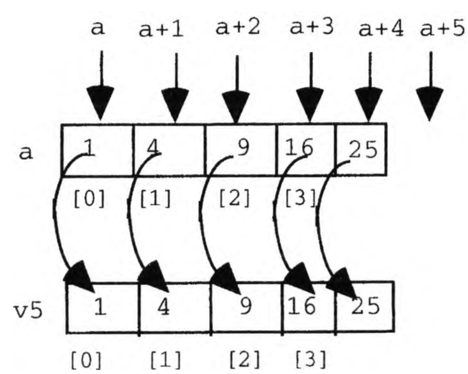
and the **`reserve()`** function increases its capacity to 7 but doesn't change its size.



The fifth declaration:

```
int a[] = {1, 4, 9, 16, 25};
vector<int> v5(a, a + 5);
```

constructs `v5` as a `vector<int>` with capacity 5 and size 5 and initializes the elements with copies of the elements of array `a`.



Some Member Functions of `vector` *to Append and Remove Values*

`v.push_back(value);`	Appends `value` at `v`'s end and increases `v`'s size by 1
`v.pop_back();`	Erases `v`'s last element and decreases `v`'s size by 1

5 In the section of `vectorlab.cpp` labeled `//--- 5 ---`, add statements to append `11` to `v2` and then output the size and contents of `v2`. Record the results below.

vector	size	contents
v2	3	0 0 11

Now add statements to append `22` to `v2` and then output the size and contents of `v2`. Record the results below.

vector	size	contents
v2	4	0 0 11 22

Now add statements to append `33` to `v2` and then output the size and contents of `v2`. Record the results below.

vector	size	contents
v2	5	0 0 11 22 33

Now add statements to erase the last element of `v2` and then output the size and contents of `v2`. Record the results below.

vector	size	contents
v2	4	0 0 11 22

Note how each `push_back()` *and* `pop_back()` *operation updates the size of the* `vector`.

`Vectors` *vs. Arrays: A Fundamental Difference*

You have seen several differences between `vectors` and arrays illustrated so far. In particular, you have examined several of the built-in operations that `vectors` have but arrays do not.

One of the most important properties of `vectors` that arrays do not possess is the ability to grow dynamically when necessary. For example, the original capacity of `v2` was 2, but you were able to append 3 new values to `v2` and it was able to accommodate each new value. This suggests that its capacity increased.

And this is indeed the case. The capacity of an array is fixed, but the capacity of a `vector` is automatically increased when necessary to accept the new values being appended. When more capacity is required, additional memory must be allocated and the current values must be copied into it. The exact implementation may vary from machine to machine, but some patterns are common to nearly all implementations. We will investigate those now.

[6] Think about an empty vector like v1. It has no capacity and so will need an increase in its capacity when the first value is appended. You should add a statement in the section of vectorlab.cpp labeled //--- 6 --- to append 111 to v1 and then output its capacity, size, and contents. Then compile and execute the program. Record your results below.

vector	size	contents
v1 after 1 value is added	1	111

[7] If you keep on adding values to v1 what happens? In the section labeled //--- 7 ---, add statements to append 222, 333, 444, and 555 to v1 and to output it's capacity, size, and contents after each value is appended. Then compile and execute the program.

v1 after	capacity	size	contents
1 value added	1	1	111
2 values added	2	2	111 222
3 values added	3	3	111 222 333
4 values added	4	4	111 222 333 444
5 values added	6	5	111 222 333 444 555

As you fill in the table you should find that even though the capacity would need only to increase by 1 to make room for a new value to be added, at some point it increases by more. The first time this happened (if any), by how much did the capacity increase? 2

[8] Now you will examine these increases in capacity by looking at a larger example. Remove the comments in the section labeled //--- 8 --- and compile and execute the program. This code adds 2495 more values to v1 and displays when the capacity changes. Record the capacity changes in the following table; include the changes from #6 and #7 above:

Size when capacity changes	0	1	2	3	4	6	9	13	19	28	42	63	94	141
v1's old capacity	0	1	2	3	4	6	9	13	19	28	42	63	94	141
v1's new capacity	1	2	3	4	6	9	13	19	28	42	63	94	141	211

Examine your results in the preceding table and see if you can formulate a rule that specifies how much a vector's capacity will increase when it is necessary to do so.

The formula seems to run around 1.5 times increase

[9] Now you should determine whether the type of elements in the vector affects the increases in capacity. Do this by adding a declaration `vector<double> v0` in the section labeled `//--- 9 ---`; and a loop similar to that used in the section labeled `//--- 8 ---`, but that runs from `i = 1` to `i = 2500`. Once you've done that, make changes in your code so that you create a `vector<char>` and do it again.

How did `vector<double>` behave?

vector<double> performs the same as vector<int>

How did `vector<char>` behave?

vector<char> performs the same as vector<int>

[10] What if the vector was originally declared with a nonzero capacity—would that make a difference? To find out, in the section of `vectorlab.cpp` labeled `//--- 10 ---`, proceed as in step 9 but use the vector `v4`, which was declared as `vector<int> v4(3, 99);`. Its capacity was later increased to 7 with the statement `v4.reserve(7);` so it has a nonzero capacity of 7. Answer the following questions:

What was the capacity of v4 before it increased automatically for the first time? 10 its size? 7

What did the capacity become when it was increased the first time? 10

The second time? 15 The third time? 22

The fourth time? 33 The fifth time? 49

What seems to be the rule? Capacity increases 1.5 times

When a `vector` is full (its size is equal to its capacity) and one or more values must be added, the capacity will be increased by some (machine-dependent) amount. Typical capacity increases are 100 percent (doubling) or 50 percent.

Member Functions to Access the First and Last Elements

`v.front();`	Returns a reference to the first value stored in v
`v.back();`	Returns a reference to the last value stored in v

Note: Since these functions return *references* to the first and last values of v, they can be used not only to retrieve the values stored in these locations but also to change them.

[11] Add a statement in the section of `vectorlab.cpp` labeled `//--- 10 ---` to output the first and last values of `v5`, which are retrieved using these function members. Compile, link, and execute the modified program and see that the output is correct. Write your statement here as well.

cout << "First val of v5 is " << v5.front() << "Last val of v5 is " << v5.back() ;

Now add statements to change the first element of `v5` to 77 and the last element to 88 using these member functions. Compile, link, and execute the modified program to see that your changes work. Then write your statements here as well.

v5.front() = 77 ; v5.back() = 88 ;

The Subscript Operator

[12.1] Look at the definition of the function `operator<<()` at the beginning of `vectorlab.cpp`. Note that it uses the subscript operator `[]` to access the `vector` elements, even though a `vector` is not an array. This must mean that `operator[]()` has been overloaded for `vectors`. Check whether this is so by replacing `v[i]` in the for loop in the definition of `operator<<()` with `v.operator[](i)` and see if it still works.

Does it? Yes

But as you will see next, the subscript operator *should be used only to access or modify values that are already in a* `vector` and not to add new values to it.

[12.2] *Note: The results of some of the following experiments may differ from one version of C++ to another.*

In the section of `vectorlab.cpp` labeled `//--- 12 ---`, try changing the element in position 1 of `v2` to 2222 using the subscript operator; i.e., use `v2[1] = 2222;`. Output the contents of `v2` to see if the change worked.

Did it? Yes

Now try appending the value 3333 to v1 with the subscript operator; that is, use `v2[v2.size()] = 3333;`. Then output `v2`.

Does it look as if 3333 got appended to `v2`? No, error in visual studio

Now enter and execute the following `for-` loop:

```
for (int i = 0; i <= v2.size(); i++)
  cout << v2[i] << "  ";
cout << endl;
```

Does the output seem to indicate that 3333 got appended to `v2`? no

Next, output the size and capacity of `v2` and record the results:

`v2.size() =` 4 `v2.capacity()=` 6

On most systems you will probably find that 3333 got inserted into the underlying array inside `v2` *after the last element of* `v2` *(if there is room for it), but the size of* `v2` *did not get updated.*

Now try one more thing with the subscript operation.

Warning: This experiment may cause a fatal run-time error, so be sure you have your files saved.

WARNING:
MAY CAUSE A
FATAL ERROR!

Try to append a value to the empty vector `v6` with the subscript operator. In other words, assign a value to `v6[0]`. Report what happens below.

Error: subscript is out of range.

Some lessons you should have learned from this experiment:

1. The subscript operator should not be used to append values to a `vector`, because neither the size nor the capacity is updated. Always use `push_back()` to append values to a `vector`. Only when a `vector` already contains values should you use the subscript operator to access or change those values.
2. As with arrays, no range checking is performed on `vectors` to ensure that indices are in range.

Some Additional Operators: Assignment, Relational, and Swap

`v1 = v2`	Assigns to `v1` a copy of `v2`
`v1 == v2`	Returns `true` if and only if `v1` contains the same values as `v2`, in the same order
`v1 < v2`	Returns `true` if and only if `v1` is lexicographically less than `v2`; that is, `v1` $\neq$ `v2`, and in the first position `i` where they differ, `v1[i] < v2[i]`
`v1.swap(v2)`	Interchanges `v1`'s contents with `v2`'s

13 Now to finish up, you will test each of these statement types. In the section of `vectorlab.cpp` labeled `//--- 13 ---`, write statements to do the following and compile, link, and execute the resulting code to observe what happens:

(a) Assign a copy of `v5` to `v3`

(b) Then check if they are equal using the `==` operator, outputting `true` if they are equal and `false` otherwise.

Record the output here: 1 (true)

(c) Check if `v5` is less than `v2` and output `true` if it is and `false` otherwise.

Record the output here: 0 (false)

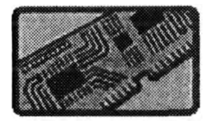
(d) Swap the contents of `v5` and `v2` and then repeat (c).

Record the output here: 1 (true)

You have finished! Hand in this lab exercise with the answers filled in, a printout of your final version of `vectorlab.cpp`, and a printout of the output it produces when executed.

Project 7.1 vectors of System Tasks

Suppose there are three different types of system tasks:



Type 1: Tasks that need only the CPU. They are identified by their:
Job-id (integer)

Type 2: Tasks that need the CPU and disk space. They are identified by their:
Job-id (integer)
Amount of disk space (integer), and
Disk drive designator (A, B, or C).

Type 3: Tasks that need the CPU and I/O devices. They are identified by their:
Job-id (integer)
Input device designator (1, 2, or 3)
Printer name (string).

Use classes for objects of types 2 and 3—and if you want, for type-1 tasks also—for which output (<<), input (>>), and less than (<) are defined. (See Note 3 below about the need for <.) You may put these classes in separate libraries or combine them in a single one.

Write a program to:

1. Define three `vectors`, one for each type of task. (Remember to `#include <vector>`.)
2. Repeatedly:
   a. Read a task (using the input operator defined for the various task types)
   b. Append the task to the `vector` for that type of task.
3. When all the data has been read, sort the contents of each `vector`, using the `sort()` algorithm from STL.

   Note 1: You must `#include <algorithm>` to use `sort()`.

   Note 2: A call to `sort()` has the form

   ```
   sort(first, beyond_last);
   ```

   where *`first`* and *`beyond_last`* are *iterators* that point to the first element in the container (`vector`) and past the last element, respectively. For `vectors`, there are two member functions that return these iterators:

`v.begin()`	Returns an iterator that points to the first element in v
`v.end()`	Returns an iterator that points to the position following the last element in v

   Note 3: `sort()` uses < (or some boolean function) to compare a container's elements. You will need to define < for tasks of types 2 and 3, where one task is less than another if the job-id of the first task is < the job-id of the second.

4. Display the number of elements in each vector and display the contents of each `vector`.

Course Info: ________________ ***Name:*** ______________________________________

Project 7.1 Grade Sheet

Hand in: Printouts showing:

1. Listings of all source files
2. A demonstration that everything compiles and links correctly
3. One or more executions of the program

Attach this grade sheet to your printouts.

Category	Points Possible	Points Received
Correctness of classes for tasks	30	________
Correctness of program (including following instructions)	70	________
Program structure (functions, etc.)	20	________
Program, class, and function documentation	20	________
Style (white space, alignment, etc.)	10	________
Total	**150**	________

Project 7.2 Building and Using a Queue Template

Background: This project consists of two parts:

Part I: *Building a Queue Template*
Part II: *Application: Queues From All Over! Enough to Drive You Crazy!!!*

Part I: *Building a Queue Template*

Convert your `Queue` class from Project 6.1 or Project 6.2 (as directed by your instructor) into a class template. It should have at least the same operations as before and the output operator (<<):

- constructor
- empty
- enqueue
- front
- dequeue
- destructor
- copy constructor
- assignment operator
- output operator (<<)

Write a driver program to test your class template.

Part II: *Queues From All Over! Enough to Drive You Crazy!!!*

Using your `Queue` template from Part I, write a program that does the following:

1. a. Defines 4 queues:
    i. `qdub1`, containing the 10 `double` values 0.0, 1.1, 2.2, 3.3, ... , 9.9
    ii. `qdub2`, containing 5 `double` values input by the user
    iii. `qdub`, an empty queue of `doubles` to be used in testing the assignment operator
    iv. `qstr`, containing a few `strings` of your choosing

   b. Outputs `qdub1` and `qdub2`.

2. Has a function template `join` that has two value (not reference or const reference) queue parameters `q1` and `q2` and that returns the queue obtained by adding the elements of `q2` at the back of `q1`. The main function should call this function with `qdub1` and `qdub2` and assign the queue returned to `qdub`,

   ```
   qdub = join(qdub1, qdub2);
   ```

   and then output the queue `qdub` that is returned.

3. Allows the user to enter any number of `Student` objects, where `Student` is a class containing a student's id number (integer) and name (string), adds each of them to a queue `qstu`, and then displays the contents of the queue. *Note:* You need not make a separate library for the class `Student`. You can just put it at the beginning of this program (by the `#includes`). It should have the input (>>) and output (<<) operators defined for it.

4. As in step 3, but use a queue `qptr` of pointers to `Students`. That is, the user input should be stored in an anonymous variable pointed to by a pointer variable and each pointer gets added to the queue. The output of the queue should display both the addresses and the contents of the memory locations pointed to by the pointers in the queue.

5. Defines a queue `qq` whose elements are the queues `qdub1`, `qdub2`, and `qdub` and outputs the contents of `qq`.

Course Info: ________________ ***Name:*** ______________________________________

Project 7.2 Grade Sheet

Hand in: Printouts that contain:

1. Listings of all source files
2. Evidence that everything compiles and links correctly
3. A sample run of the binary executables produced

Attach this grade sheet to your printouts.

Category	Points Possible	Points Received
Part I: Queue Class Template and Driver Program:		
Correctness of template mechanism	25	____________
Queue operations	25	____________
Documentation	5	____________
Style (white space, alignment, etc.)	5	____________
Driver adequately tests the class template	20	____________
Total for Part I	**80**	____________
Part II: Application: Queues From All Over		
1.	20	____________
2.	20	____________
3.	20	____________
4.	20	____________
5.	20	____________
Total for Part II	**100**	____________