Lab #2
Dynamically Growing Arrays

Introduction

In this lab you will implement a dynamically growing array, also referred to as a *vector*, a common data structure used to store a set of elements that can significantly vary in number throughout the execution of a program. When a vector is created, an initial region of memory is assigned to it. As the first few elements are inserted into the vector, this memory region is progressively occupied, until eventually it fills up.

When a new element is inserted at this moment, the capacity of the vector must be increased in order to make additional room. Increasing the capacity of the vector involves reallocating its memory, copying the content of the original vector into the new vector, and freeing the old vector. This is a costly process, and thus, we should avoid repeating it upon every subsequent element insertion. Instead, we will add room for multiple additional elements every time the vector capacity grows. Our choice will be doubling the vector capacity every time we exceed the current storage limit.

A vector of double-precision floating-point numbers can be represented with the following three global variables:

- `double *v`. This is a pointer to the first element in a sequence of elements appearing consecutively in memory.

- `int count`. This variable represents the number of elements inserted in the array so far by the user. Its initial value is 0.

- `int size`. This variable represents the space currently allocated for the array, given in number of elements. We will initialize this value to 2.

The main program

In order to be able to test our program right away, we will start by writing code for an interactive main program that asks the user to select an element from a menu containing a set of possible actions. The application should display the following message:

```
Main menu:

1. Print the array
2. Append element at the end
3. Remove last element
4. Insert one element
5. Exit

Select an option: _
```

Similarly, the main program should invoke functions `Initialize()` when the program begins, and `Finalize()` when it ends. The former will initialize the three global variables associated with the vector with valid values, and allocate memory for the vector with an initial capacity of just 2 elements. The latter will free the memory associated with the vector.

Pre-lab Assignment (2 pt.)

To complete this pre-lab assignment, use the CoE Linux machines either locally at 271 Snell, or remotely through an SSH client. At the beginning of the lab session, the Teaching Assistants will ask you to connect to these machines and show a working version of your program. You will be asked to show the code and to explain it. Both members of the team should be able to answer questions individually.

Write a program that displays the menu shown above, and waits for the user to enter an option. If the option is invalid, an error message should be displayed and the main menu should be shown again. When the user selects option 5, the program should finish. When the user enters any other valid option, your pre-lab program should just print the name of the option on the screen. We will replace the code for each option with its actual functionality in the following assignments. For example:

```
      Select an option: 4
      You selected “Insert one element”
```

After a valid option is selected and the proper message is displayed, the main menu should be displayed again. Use a `switch` control flow statement to manage the menu.

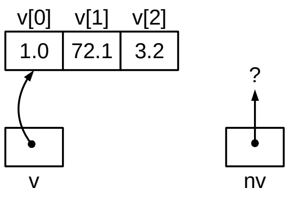
a) (1 pt.) Upload the code for this pre-lab assignment on Blackboard on a file named `prelab.cc`. This program should contain functions `main()`, `Initialize()`, and `Finalize()`. The body of functions `Initialize()` and `Finalize()` should be empty for now—we will populate them in the following assignments.

b) (1 pt.) Run your program and show the output of an execution example where you select several different options. You can copy and past the terminal output into your report, or you can add screenshots.

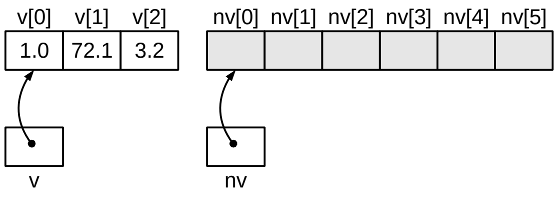
Growing the vector

You will now work on a function that grows the capacity of the vector. This function should increase the vector's allocated storage while keeping the same set of elements in it, using the following steps:

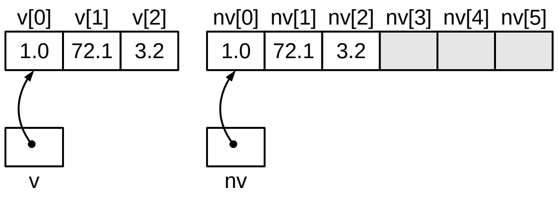
- Initially, we have a vector `v` that has reached its full capacity, and an uninitialized pointer `nv` to what will be the new vector.



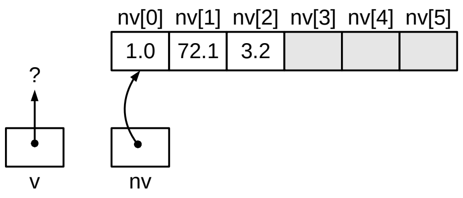
- We first allocate a new memory region that will serve as the new storage for the vector, with the desired new capacity.



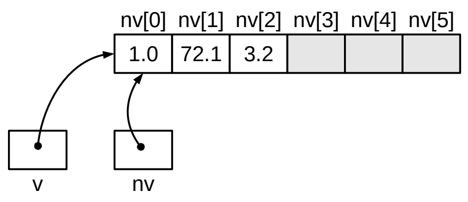
- All elements in the old vector are copied into the new vector. The additional elements in the new vector remain uninitialized.



- The memory originally allocated for old vector is now freed.



- We make the old vector pointer `v` point to the new memory region referenced by `nv`. We can now continue to use `v` normally for future insertion, deletion, search, or update operations.



Assignment 1 (2 pt.)

Show the new code for function `Grow()`. The function should print debug information on the screen, including the old and new capacities of the vector. For example:

```
Vector grown
Previous capacity: 2 elements
New capacity: 4 elements
```

Adding an element at the end

Adding an element at the end of the vector may require to grow its capacity. We will do so only when the current number of present elements is equal to the capacity of the vector, by invoking function `Grow()`. Once we make sure that there is enough storage capacity for the new element, a new element can be safely inserted at the end of the vector.

Assignment 2 (2 pt.)

Write function `AddElement()`. The body of this function should ask the user to enter a floating-point number, which will be the new value added at the end of the vector.

```
    Enter the new element: _
```

Write another function named `PrintVector()`, which should display the current content of the vector. The functionality to add an element and to print the vector should be now available through options 2 and 1 in the menu, respectively.

a) (1 pt.) Show the code for functions `AddElement()` and `PrintVector()` on your report.

b) (1 pt.) Include the output of your program for an execution where you add several elements (enough to make the vector grow at least once), and then display its current content. You can copy and paste the output of the terminal or simply add a screenshot.

Removing an element from the end

This action is accessible through option 3 in the main menu. If the vector is empty and the user selects this option, a proper error message should be displayed, indicating that there are no elements left to remove.

Assignment 3 (2 pt.)

Write function `RemoveElement()` with the described functionality.

a) (1 pt.) List the code of this function in your report.

b) (1 pt.) Show the output of the program for two execution scenarios: one where removing the last element happens successfully, and one where the function is invoked on an empty vector.

Inserting an element

In general, inserting an element at a random position of the vector involves shifting all elements that appear on its right before the actual insertion, in order to make room for the new element. It also involves growing the vector in advance, in the case that the previous number of elements is equal to the capacity of the vector.

A generic insertion operation requires two pieces of information: the index that will be occupied by the new element, and the element to be inserted itself. These two values need to be requested from the user. Notice that the valid range for the index is between 0 and *n*, where *n* is the current number of elements. Passing a value of *n* to the index is equivalent to adding the new element at the end of the vector.

Assignment 4 (2 pt.)

Write function `InsertElement()` and make it accessible through option 4 on the menu. The function should ask the user for an index and a value for the new element. The index should be checked for correct boundaries, and a proper error message should be displayed if the entered value is invalid.

```
Enter the index of new element: _
Enter the new element: _
```

a) (1 pt.) List the code for function `InsertElement()` in your report.

b) (1 pt.) Show the output of your program for an execution where you enter several elements at different intermediate positions. Cover the case where you enter an invalid value for the index.

Shrinking the vector

Suppose that an application needs to insert a large amount of elements into a vector during its initialization phase, but most of those elements are then released, leaving the vector with a small load for the rest of the execution. To reduce the application's memory footprint, it would be reasonable to shrink the memory space allocated by the vector as soon as its occupancy is reduced under a certain threshold.

Extra Credit (2 pt.)

Write a function named `Shrink()` that reallocates the vector with half of its original capacity, maintaining its content intact. Have this function dump debug information related with the previous and new capacity of the vector. Invoke the function when the user deletes elements from the vector and its occupancy becomes lower than 30% of the total capacity. Show a program execution example where the correct behavior of function `Shrink()` is demonstrated.