Project 4 — Vector of int

CS 2370

Background

You have been using **std::vector** for a while now. In this assignment you will implement a simple version of vector, that just holds **int**s. You will define some of the same member functions that **std::vector** does, and they will behave the same way. The goal of this project is to learn how to manage *dynamic memory* on the heap. You will allocate an array on the heap for your Vector class to store **int**s in. When your underlying memory is used up, and you want to append or insert a new element, you will need to allocate a new array, copy the current elements to it, and then delete the old array.

Requirements

Call your class **Vector** (capital-v). Your constructor should allocate an initial array big enough to hold 10 ints (a constant named CHUNK), which is its initial "capacity", but the "size" (the numbers of elements currently in use) will initially be zero. Do not hard-code the 10. (See the header below). When you expand the array, make the new capacity 1.6 times the current capacity (initially CHUNK). The following diagram illustrates a **Vector** instance managing heap memory.

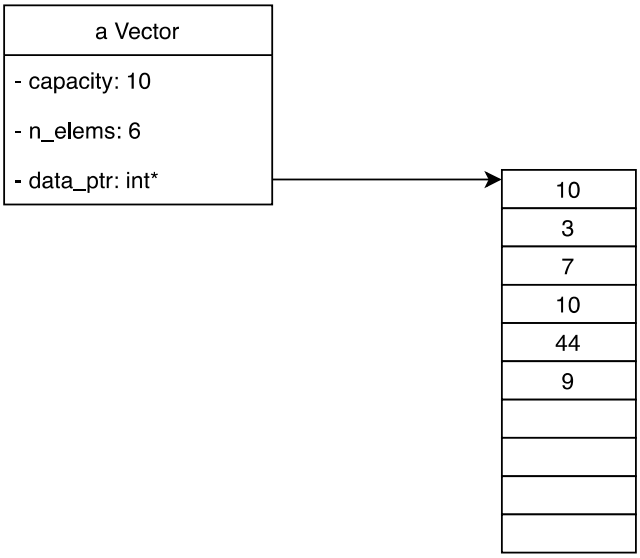

Figure 1 How Vector uses a Dynamic Array

I am providing the **Vector.h** file for you that declares everything in the class. Your job is to implement the member functions in the file **Vector.cpp**. The file **Vector.h** is found in the Canvas assignment, along with a test driver for it.

Here is the **.h** file:

```
#ifndef VECTOR_H
#define VECTOR_H
```

```
#include <cstddef>
using std::size_t;

class Vector {
   enum {CHUNK = 10};
   int* data_ptr;     // Pointer to the heap array
   size_t capacity;   // Size of the current array allocation (total number of ints)
   size_t n_elems;    // Number of int spaces currently in use, starting from pos 0
   void grow();
public:
   // Object Mgt.
   Vector();
   Vector(const Vector& v);          // Copy constructor
   Vector& operator=(const Vector& v); // Copy assignment operator
   ~Vector();

   // Accessors
   int front() const;              // Return the int in position 0, if any
   int back() const;               // Return last element (position n_elems-1)
   int at(size_t pos) const;       // Return element in position "pos" (0-based)
   size_t size() const;            // Return n_elems
   bool empty() const;             // Return n_elems == 0

   // Mutators
   int& operator[](size_t pos);      // Same as at but no bounds checking
   void push_back(int item);         // Append a new element at the end
   void pop_back();                // --n_elems (nothing else to do)
   void erase(size_t pos);           // Remove item in position pos by shuffling left
   void insert(size_t pos, int item);  // Shuffle items right to make room for new element
   void clear();                   // n_elems = 0 (keep the current capacity)

   // Iterators
   int* begin();  // Return a pointer to 1st element, or nullptr if n_elems == 0
   int* end();          // Return a pointer to 1 past last element, or nullptr if n_elems == 0

   // Comparators
   bool operator==(const Vector& v) const;
   bool operator!=(const Vector& v) const;
};

#endif
```

In this project we are introducing the **size_t** type synonym, defined in **<cstddef>**. **size_t** is used for array/vector index positions and for the size (number of elements) of allocated memory. It is *synonym* because it is equivalent to either an **unsigned int** or an **unsigned long**, depending on the platform. We use it whenever we refer to array *sizes* or *positions*.

You are also provided a test driver, **testVector.cpp**, and a test framework in **test.h**, to rigorously test your project before you turn it in. The expected output is:

```
grow

Test Report:

    Number of Passes = 78
    Number of Failures = 0
```

The occurrence of “grow” above is a trace statement from your **grow** function, to verify that it executed once. It is best to comment-out much of the testing while you are developing your solution, so you test a little at a time. Your grade will be the percentage of tests that you pass.

Implementation Notes

I have given you the hint that will need a private grow function whenever you need to expand the array from the **push_back** or **insert** functions.

In all cases where an array position is mentioned, check if it is within bounds, except for in the **operator[]** function. Throw a **std::range_error** (defined in **<stdexcept>**) in that case. This includes **front** and **back** as well, since they fail when the size is 0. Six functions need to throw exceptions. Include an informational inside the exception object.

Remember that you will be doing dynamic memory allocation in the copy constructor, copy assignment operator and copy assignment operator. The destructor frees the heap memory with the **delete** operator.