CSCI 152, Performance and Data Structures, Assignment 1

Rules for Assignments

- You are expected to read the complete assignment and to complete every part of it. 'I did not see this part' is never a valid excuse for not completing a part of the assignment.
- Submitted code will be checked for correctness, correct memory management, readability, style, and layout. We use **valgrind** for memory checks. We apply high quality standards during grading.
- Assignments are graded with the help of autograders. Make sure that your submitted code can be compiled. If you don't know how to complete a task, insert an empty function, so that your code compiles in all cases.
- In order to make it possible to adopt your code to our automatic checkers, and to give more feedback, we check your code at least twice. The last submitted version counts. Use the earlier checks. If you submit only for the last check, you are taking unnecessary risk.
- If you need help, you can either **(1)** come to a lab session, or **(2)** post your question on Piazza in a private question (with visibility set to 'all instructors'.) Do not mail directly to an instructor.
- Don't wait until the last moment with starting to work on the assignment.
- You must write all submitted code by yourself! Even allowing others to see your code, or looking at others' code is already academic misconduct.

C^{++} Coding Rules

- Avoid unitialized variables. If you don't have a value to initialize a variable, you are almost certainly declaring it too early. C^{++} allows declarations of variables almost everywhere. Declare variables where you need them for the first time.
- Make sure to use `size_t` for all indexing. (Not `int`, nor `unsigned int`.) Don't write character contants by their ASCII codes. (Never write 'a' instead of 97.)

- Do not implement anything in header files. In real life, short methods should be implemented in header files, while longer methods should be implemented in **.cpp** files. Because of the way that we test, this is not possible in assignments. We test with our own header files, so everything that you write in a header file, will be ignored.
- You are not allowed to use any STL containers or library functions, unless explicitly mentioned in the assignment. If in doubt, ask in Piazza.
- Don't use `printf`, `malloc`, `free`, or `memcpy`. Don't use `#define`, except in include guards.
- Don't use `0` or `NULL` to represent the null-pointer. The only correct representation of the null-pointer is `nullptr`. It is OK to write `if(p)` if you want to test that `p` is not the null-pointer.
- Avoid `break` and `continue`. They are just `goto` in disguise. In nearly all cases, they can be avoided by changing the condition of the **while** loop, which always results in more readable code.
- Don't `throw` and `catch` an exception inside the same function. That is not what exceptions are intended for.
- Avoid assignments in constructor bodies. Use member initializers wherever possible.
- Don't write: `if(b) return true; else false;` Just write `return b;`

1 Introduction

Goal of this assignment is to make you familiar with the implementation of containers that use the heap in C^{++}. This will be the basis for implementing data structures in later assignments. In C^{++}, memory management is semi-automatic. This means that the implementer of a type has to provide a couple of functions that take care of the memory management, but the compiler will automatically insert these functions where needed.

We will reimplement `std::vector` as a `csci152::vector`. A vector is similar to an array, but it stores its main data on the heap, and grows or shrinks dynamically when needed. Whenever the vector runs out of space, it will reallocate its memory, always doubling its size. Always allocating exactly what is needed would be so inefficient that it would make the vector useless for any applications. In order to make sure that the use of the heap is invisible from outside, you will need to implement the constructors, copying assignment, and the destructor of the vector class.

The implementation of `csci152::vector` is based on the class definition below. We use `sz` for the current size, and `cap` for the current capacity. `heap` is a pointer to the allocated heap memory. When the vector runs out of capacity (this happens when sz == cap and more elements need to be pushed), it reallocates heap, and doubles cap. We will insist that cap is always a power of 2. The vector is declared in such a way that it is easy to change the type of the elements in the vector. For this purpose, we added using basetype = double, and don't further refer to double in the rest of the implementation. During some of your tests, you should replace double by std::string. If you did everything correctly, the vector will also work for strings.

In the definition below, all fields are public, but they shouldn't be public in a real implementation. Usually one makes them public during testing, and private when the implementation is finished. Since you will be testing, they are public.

```
namespace csci152
{
   using basetype = std::string;
      // Must be default constructible, copyable, and assignable.
      // In your implementation, only refer to basetype.

   class vector
   {
      size_t sz;
      size_t cap;     // Always power of two. We always have sz <= cap.
      basetype* heap; // Different vectors do not share heap memory.

   private:
   public:    // You can delete this line later.

      static size_t minpow2( size_t x );
         // Return smallest power of 2 >= x,

      void reallocate( size_t c );
         // Reallocate to the smallest power of two <= c.
         // This function always reallocates, so one must
         // think before one calls it.

    ...
   };
}
```

2 Tasks

Download the files **vector.h**, **vector.cpp**, **main.cpp**. Also download the **Makefile**.

1. Write the method size_t minpow2(size_t x). This function must return the smallest power of two that is greater or equal to x. In order to

call it from outside, write `csci152::vector::minpow2( )`.

2. Implement the default constructor `vector( )`. Try to use member initialization as much as possible, and make sure to write the fields in the order in which they appear in the class definition. If you write them in a different order, they will still be initialized in the order in which they appear in the class definition, and the compiler will warn you about this. The constructor should allocate one `basetype`.

3. Implement the copy constructor `vector( const vector& other )`. Use `minpow2` to decide how much space must be allocated. It is possible that `other` has reserved much more space than needed, and we don't want to copy that.

4. Implement the `initializer_list` constructor `vector( std::initializer_list< basetype > values )`. Again use `minpow2` to determine how much space must be allocated. In order to access the elements in an `initializer_list`, use a range-for. With this constructor you can write for example `csci152::vector vect = { 1,2,3,4};`

5. Implement `void print( std::ostream& out ) const;` This function must print the elements of the vector, between `[` and `]`, and separated by commas. We provided an implementation of `operator <<`, that calls `print( )`.

   ```
   csci152::vector vect1 = { 1,2,3,4 };
   std::cout << vect1 << "\n";
      // prints [ 1, 2, 3, 4 ]
   csci152::vector vect2 = { 100 };
   std::cout << vect2 << "\n";
      // prints [ 100 ]
   ```

6. Implement `void reallocate( size_t c )`. This function **always** reallocates to the smallest power of two that is greater or equal to `c`.

7. Implement the destructor `~vector( );`

8. Implement the assignment operator `vector& operator = ( const vector& other );` Make sure to check for self-assignment. Check that `cap` is big enough to contain `other.sz`. If it is not, then call reallocate. You don't have to reallocate if `other.sz` is smaller than `cap`.

   Now you have fulfilled the rule of 3. Test carefully with **valgrind**.

9. Implement `void push_back( const basetype& val )`, which appends `val` to the end of the vector. Check if there is room for `val`. If not, then call `reallocate`.

10. You can now write the indexing methods. In C^{++}, indexing comes in two variants, *checked* and *unchecked*. Simple indexing with `operator[]` is unchecked, which means that trying to access an element at a position `>= size( )` is UB (*undefined behaviour*). Indexing with the `at` method is checked. This means that trying to access an element at a position `>= size( )` will result in an `std::out_of_range` exception.

    In addition to checked or unchecked, indexing methods can also be **const** or **non-const**. In contexts where the vector can be changed, the compiler will use the **non-const** versions which can give out a **non-const** reference to the contents of the vector. Otherwise, a **const** reference is given.

    ```
    basetype& operator[]( size_t index );
    const basetype& operator[]( size_t index ) const;

    basetype& at( size_t index );
    const basetype& at( size_t index ) const;
    ```

11. Write

    ```
    basetype& front( );
    const basetype& front( ) const;

    basetype& back( );
    const basetype& back( ) const;
    ```

    All of these methods cannot be called when the vector is empty. You don't need to check that. If you feel bad about that, you can throw an `std::out_of_range` exception.

12. Implement method `void pop_back( )`. This method removes the last element from the vector. The method cannot be called when the vector is empty. Either don't check this, or throw `std::out_of_range` when the vector is empty. Checking and ignoring is not acceptable. You don't need to reallocate if `sz` becomes much smaller than `cap`.

13. Implement method `void reserve( size_t c )`. This method ensures that `cap` is at least `c`. (But always a power of 2.)

14. Implement method `shrink( )`. This method makes sure that the capacity is just enough to hold the current size. (But still a power of 2.)

15. Implement method `void insert( size_t i, const basetype& val )`, which inserts `val` at position `i`. This operation can have UB if `i > size( )`. Check if there is room for `val` and reallocate if needed.

    Note that `val` must be not in the vector. Calling `vect. insert( 0, vect[1] )` is asking for trouble (UB).

16. Implement method `void erase( size_t i )`, which erases the `i`-th element from the vector by moving the elements behind it one position to the front. This operator may have UB if `i >= size( )`.

Now your implementation of `vector` should be complete. If you did everything well, `vector` of `double` can be changed into `vector` of `string`, by changing `basetype = double` into `basetype = std::string`.

The `main` function will not compile with strings. Change `#if 1` at the beginning to `#if 0` and the second part after `#else` will be used.

In general, you have to create tests for your code by yourself. The file **main.cpp** contains a few tests, but you have to create more tests by yourself. We will use different (and stricter) tests during grading. Make sure that all implemented methods are called in your tests, including copy-constructors and assignment. Always check for self-assignment, if you implement an assignment method.

Test on Linux with `valgrind`. Check the compiler warnings. It is useful to test with compiler options `-flto -O3`, because it reveals more problems.

3 How to Submit

Submit two files **vector.cpp** and **main.cpp**. Don't use an archiver. Make sure that your code runs with the original **vector.h**, but also test with `basetype = std::string;`