EEE2020-03 **Spring 2021**

Assignment 1: Vector (Array List)

Due: Sunday, Apr. 18, 2021, 11:59PM

1 Introduction

- The objective of this assignment is to implement a vector class (a.k.a array list ADT).
- In particular, you will have to complete the missing parts of `class vector_t` based on the lecture notes of `3_lists.pdf`.
- In Ubuntu or Mac terminal, use `wget` to download a copy of skeleton code compressed into `vector.tar`.

```
$ wget https://icsl.yonsei.ac.kr/wp-content/uploads/vector.tar
```

- Decompress the `tar` file, and go to the `vector/` directory. Then, type `ls` to find the following list of files.

```
$ tar xf vector.tar
$ cd vector/
$ ls
data.h  main.cc  Makefile  tar.sh  vector.cc  vector.h
```

- From the listed files, `vector.cc` is the only file you will have to work on.
- You are allowed to change other files for further testing and validation, but they will revert to the original state when your assignment is graded.
- To compile the code, you can simply type `make` in the terminal. `Makefile` that comes along with the skeleton code has automated all the compiling scripts for you.

```
$ make
g++ -Wall -Werror -g -o main.o -c main.cc
g++ -Wall -Werror -g -o vector.o -c vector.cc
g++ -o vector main.o vector.o
```

- Executing the program should print the following lines.

```
$ ./vector
capacity=0
size=0
array=[ ]

capacity=0
size=0
array=[ ]

capacity=0
size=0
array=[ ]

capacity=0
size=0
array=[ ]

capacity=0
```

```
size=0
array=[ ]

capacity=0
size=0
array=[ ]

capacity=0
size=0
array=[ ]
```

- To clean up the directory and rebuild the program, you can type `make clean` and then `make` again.

```
$ make clean
rm -f main.o vector.o vector

$ make
g++ -Wall -Werror -g -o main.o -c main.cc
g++ -Wall -Werror -g -o vector.o -c vector.cc
g++ -o vector main.o vector.o

$ ./vector
capacity=0
size=0
array=[ ]

...
```

2 Implementation

- In `vector.h`, you will find the definition of `class vector_t`. The header file is complete, and you do not have to touch anything in this file. Take a look at how `class vector_t` is defined - *"can you reproduce this if you were to start the assignment from scratch?"*

```
/* vector.h */

#ifndef __VECTOR_H__
#define __VECTOR_H__

#include "data.h"

class vector_t {
public:
    vector_t(void);
    vector_t(const vector_t &m_vector);
    ~vector_t(void);

    // Get the number of elements in the array.
    size_t size(void) const;
    // Get the allocated array size.
    size_t capacity(void) const;
    // Get a pointer to the first element in the array.
    data_t* begin(void) const;
    // Get a pointer to the next of last element.
    data_t* end(void) const;
    // Get a reference of element at the given index.
    data_t& operator[](const size_t m_index) const;
```

```
    // Assign new contents to the vector, and replace the existing array.
    vector_t& operator=(const vector_t &m_vector);
    // Allocate a memory space of the given size, if it greater than the current.
    void reserve(const size_t m_array_size);
    // Remove all elements, but keep the array.
    void clear(void);
    // Add a new element at the end of array.
    void push_back(const data_t m_data);
    // Remove the last element in the array.
    void pop_back(void);
    // Insert a new element at the given location.
    void insert(data_t *m_ptr, const data_t m_data);
    // Remove an element at the given location.
    void erase(data_t *m_ptr);

private:
    data_t *array;          // Pointer to an array
    size_t array_size;      // Allocated array size
    size_t num_elements;    // Number of elements in the array
};

#endif
```

- The next shows the contents of `vector.cc` that has the actual implementation of `class vector_t`.
- In the source file, class constructor, copy constructor, and destructor are already implemented. In addition, several simple functions such as `size()`, `capacity()`, `begin()`, `end()`, and `operator[]` are also provided. You do not have to touch them.
- For your reference, `operator=` is also implemented in the skeleton code. Have a close look at this function to get ideas about how to implement other missing functions.
- It must be obvious in the skeleton code to find which functions are incomplete. The function bodies are marked with `/* Assignment */`. Fill in the functions to complete this assignment.

```
/* vector.cc */

#include <cstdlib>
#include <new>
#include "vector.h"

// Constructor
vector_t::vector_t(void) :
    array(0),
    array_size(0),
    num_elements(0) {
    // Nothing to do
}

// Copy constructor
vector_t::vector_t(const vector_t &m_vector) :
    array_size(m_vector.num_elements),
    num_elements(m_vector.num_elements) {
    // The array is a tight fit.
    array = (data_t*)malloc(sizeof(data_t) * array_size);
    // Copy data.
    for(size_t i = 0; i < num_elements; i++) {
        new (&array[i]) data_t(m_vector.array[i]);
    }
```

```
}

// Destructor
vector_t::~vector_t() {
    // Destruct all data.
    for(size_t i = 0; i < num_elements; i++) { array[i].~data_t(); }
    // Deallocate the array.
    free(array);
}

// Get the number of elements in the array.
size_t vector_t::size(void) const {
    return num_elements;
}

// Get the allocated array size.
size_t vector_t::capacity(void) const {
    return array_size;
}

// Get a pointer to the first element in the array.
data_t* vector_t::begin(void) const {
    return array;
}

// Get a pointer to the next of last element.
data_t* vector_t::end(void) const {
    return array + num_elements;
}

// Get a reference of element at the given index.
data_t& vector_t::operator[](const size_t m_index) const {
    return array[m_index];
}

// Assign new contents to the vector, and replace the existing array.
vector_t& vector_t::operator=(const vector_t &m_vector) {
    if(&m_vector != this) {
        // Destruct all existing data elements.
        for(size_t i = 0; i < num_elements; i++) { array[i].~data_t(); }
        // Resize the array if the new array is bigger than the current one.
        if(array_size < m_vector.num_elements) {
            free(array);
            // The new array is a tight fit.
            array_size = m_vector.num_elements;
            array = (data_t*)malloc(sizeof(data_t) * array_size);
        }
        // Copy data.
        num_elements = m_vector.num_elements;
        for(size_t i = 0; i < num_elements; i++) {
            new (&array[i]) data_t(m_vector.array[i]);
        }
    }
    return *this;
}
```

```
/************************
 * EEE2020: Assignment 2 *
 ************************/

// Allocate a memory space of the given size, if it is greater than the current.
void vector_t::reserve(const size_t m_array_size) {
    /* Assignment */
}

// Remove all elements, but keep the array.
void vector_t::clear(void) {
    /* Assignment */
}

// Add a new element at the end of array.
void vector_t::push_back(const data_t m_data) {
    /* Assignment */
}

// Remove the last element in the array.
void vector_t::pop_back(void) {
    /* Assignment */
}

// Insert a new element at the given location.
void vector_t::insert(data_t *m_ptr, const data_t m_data) {
    /* Assignment */
}

// Remove an element at the given location.
void vector_t::erase(data_t *m_ptr) {
    /* Assignment */
}

/*********************
 * End of Assignment *
 *********************/
```

- The first incomplete function in `vector.cc` is `reserve()`. This function is supposed to allocate a memory space of the given size (i.e., `m_array_size`), only if the requested size is greater than the current array size. Otherwise, the function does nothing. When a new memory space has been allocated, all the existing data in the previous memory block must be copied to the new memory space. Then, the old memory block is deallocated. Since `data_t` is simply `unsigned int` in this assignment, you do not have to worry about calling copy constructors and destructors of `data_t`.
- `clear()` removes all elements in the array, but it keeps the array space. Since there is nothing to destruct for `int`, this function can be as simple as setting `num_elements = 0`.
- `push_back()` adds a new element at the end of array. If the array is full before adding the new element, the array size should double up. You may call `reserve(2 * array_size)` to let the `reserve()` function handle resizing the array. If no array has been previously allocated (i.e., `array_size == 0`), an one-element array is created to store the new element.
- `pop_back()` removes the last element in the array. The array does not shrink after removing the element. Since there is nothing to destruct for `int`, this function can be as simple as `--num_elements`. However, be aware that the function can possibly be called when the array is empty and has nothing to remove.
- `insert()` adds a new element at the specified position pointed by a pointer (i.e., `m_ptr`). This function does not check if the pointed memory address is valid, whether it falls into the valid array region. When such an

erroneous access is made, the code will simply throw a `segmentation fault` while running. It is not the responsibility of array list ADT to catch all errors, but it is rather programmers' responsibility to correctly use the array list ADT in `main()` or any caller functions. However, `m_ptr` being `end()` is allowed for `insert()`, since the operation then be identical to `push_back()` that adds a new element at an unoccupied slot at the end.

- `erase()` removes an element at the specified memory location pointed by a pointer (i.e., `m_ptr`). Again, it does not check if the pointed memory address is a valid one. For an erroneous access, executing the code will simply throw a `segmentation fault`. Unlike `insert()`, `m_ptr` being `end()` is not a valid operation because `end()` points to an invalid element.
- Lastly, `main()` looks as follows, and it tests various functions of `class vector_t`. Your are allowed to change `main()` to further test and validate your code, but note that this file will revert to the original state when your assignment is graded.

```
#include <iostream>
#include "vector.h"

using namespace std;

// Print the information of vector_t.
void print(const vector_t &m_vector) {
    cout << "capacity=" << m_vector.capacity()  << endl
         << "size="     << m_vector.size()      << endl
         << "array=[ ";
    for(size_t i = 0; i < m_vector.size(); i++) {
        cout << m_vector[i] << " ";
    }
    cout << "]"                          << endl << endl;
}

int main(void) {
    // Create a vector_t variable named vec.
    vector_t vec;

    // Test push_back().
    vec.push_back(17);
    vec.push_back(8);
    vec.push_back(11);
    vec.push_back(82);
    vec.push_back(179);
    vec.push_back(41);
    print(vec);

    // Test insert().
    vec.insert(vec.begin(), 2);
    vec.insert(vec.begin()+1, 27);
    vec.insert(vec.begin()+4, 19);
    vec.insert(vec.end(), 35);
    vec.insert(vec.end()-1, 94);
    vec.insert(vec.end()-3, 101);
    print(vec);

    // Test pop_back().
    vec.pop_back();
    vec.pop_back();
    vec.pop_back();
    print(vec);

    // Test erase().
```

```
    vec.erase(vec.begin());
    vec.erase(vec.begin()+2);
    vec.erase(vec.end()-1);
    vec.erase(vec.end()-3);
    print(vec);

    // Create a vector_t variable named cp as a copy of vec.
    vector_t cp = vec;
    print(cp);

    // Test clear() and assignment operator.
    cp.clear();
    vec = cp;
    print(vec);

    // Test reserve(), push_back(), insert(), and assignment operator.
    cp.reserve(10);
    vec.push_back(57);
    vec.insert(vec.begin()+1, 90);
    vec.insert(vec.end()-2, 4);
    cp = vec;
    print(cp);

    return 0;
}
```

- If `class vector_t` is correctly implemented, executing the code will produce the following output.

```
$ ./vector
apacity=8
size=6
array=[ 17 8 11 82 179 41 ]

capacity=16
size=12
array=[ 2 27 17 8 19 11 82 179 101 41 94 35 ]

capacity=16
size=9
array=[ 2 27 17 8 19 11 82 179 101 ]

capacity=16
size=5
array=[ 27 17 19 82 179 ]

capacity=5
size=5
array=[ 27 17 19 82 179 ]

capacity=16
size=0
array=[ ]

capacity=10
size=3
array=[ 4 57 90 ]
```

- Even if your code compiles and runs to completion with correct results, you need to double-check the code via

`valgrind` to confirm no memory leaks.

```
$ valgrind ./vector

...

==40205== All heap blocks were freed -- no leaks are possible

...
```

- In Mac OS, `valgrind` is not well supported. You may use `leaks` instead to test memory leaks.

```
$ leaks --atExit -- ./vector

...

Process 31661: 0 leaks for 0 total leaked bytes.

...
```

3 Submission

- When the assignment is done, execute the `tar.sh` script in the `vector/` directory.
- It will compress the `vector/` directory into a `tar` file named after your student ID such as `2020140000.tar`.

```
$ ./tar.sh
rm -f main.o vector.o vector
a vector
a vector/main.cc
a vector/data.h
a vector/Makefile
a vector/vector.h
a vector/tar.sh
a vector/vector.cc

$ ls
2020140000.tar  data.h  main.cc  Makefile  tar.sh  vector.cc  vector.h
```

- Upload the tar file (e.g., `2020140000.tar`) on LearnUs. Do not rename the `tar` file or C++ files included in it.

4 Grading Rules

- The following is the general guideline for grading. 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change, and the grader may add a few extra rules for fair evaluation of students' efforts.

  -3 points: A submitted `tar` file is renamed and includes redundant tags such as `project2`, student name, etc.

  -5 points: A program code does not have sufficient amount of comments. Comments in the skeleton code do not count. You must make an effort to clearly explain what each part of your code intends to do.

  -5 points each: There are total seven sets of printed blocks (i.e., `capacity= ..., size= ..., array=[ ... ]`). For each incorrect block, 5 points will be deducted.

  -15 points: The code compiles and runs, but it has memory leaks or crashes at the end with errors.

  -25 points: The code does not compile or fails to run (e.g., no or completely wrong results), but it shows substantial efforts to complete the assignment.

-30 points: No or late submission. The following cases will also be regarded as no submissions.

* Little to no efforts in the code (e.g., submitting nearly the same version of code as the skeleton code) will be regarded as no submission. Even if the code is incomplete, you must make substantial amount of efforts to earn partial credits.
* Fake codes will be regarded as cheating attempts and thus not graded. Examples of fake codes are i) hard-coding a program to print expected outputs to deceive the grader as if the program is correctly running, ii) copying and pasting random stuff found on the Internet to make the code look as if some efforts are made. More serious penalties may be considered if students abuse the grading rules.

Final grade = F: The submitted code is copied from someone else. All students involved in the incident will be penalized and given F for the final grades irrespective of assignments, attendance, etc.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect for any reasons, feel free to discuss your concerns with the TA. In case no agreement is made between you and the TA, elevate the case to the instructor to review your assignment. Refer to the course website for the contact information of TA and instructor: `https://icsl.yonsei.ac.kr/eee2020`
- Begging partial credits for no valid reasons will be treated as a cheating attempt, and such a student will lose all scores of the assignment.