Name: Le Pham Nhat Quynh
SID: 20125110

REPORT

I. String class:

1. Definition:

Strings are character sequences that are represented as objects. The standard string class supports such objects by providing an interface that is comparable to that of a regular container of bytes, but with additional characteristics tailored to single-byte character strings.

```
class string {
   char* str;

public:
   string();
   string(char* val);
   string(const string& source);
   ~string();
   // etc
};
```

2. Properties:

Internally, the C++ string class utilizes a char array to hold characters, but the string class handles all memory management, allocation, and null termination, which is why it is so simple to use. Because of dynamic memory allocation, similar to vectors, the length of a C++ string can be altered at runtime. Because the string class is a container class, we may use an iterator to traverse over all of its characters, much like other containers like vector, set, and maps. However, we usually use a simple for loop to iterate over the characters and index them with the [] operator.

The C++ string class has a number of methods that make it simple to work with strings. The most useful ones are shown in the code below.

3. Public member function:

a) Constructor:

- Default: string(); - Creates an empty string with no characters in it.

```
string::string(): str{ nullptr }
{
   str = new char[1];
   str[0] = '\0';
}
```

- Copy: string (const string& source); - Creates a duplicate of str.

```
string::string(const string& source)
{
   str = new char[strlen(source.str) + 1];
   strcpy(str, source.str);
   str[strlen(source.str)] = '\0';
}
```

- From c-string: string (const char* s); - Copies the s-pointed null-terminated character sequence.

```
string::string(const char* s)
{
   if (!s)
   {
      str = new char[1];
      str[0] = '\0';
   }
   else
   {
      str = new char[strlen(s) + 1];
      strcpy(str, s);
      str[strlen(s)] = '\0';
   }
}
```

b) Destructor:

- ~string - This frees up all of the storage space allocated by the string's allocator.

```
string::~string()
{
      delete str;
}
```

c) Capacity:

- length(): Returns the length of the string

```
int string::length()
{
   return strlen(str);
}
```

- emty(): Returns True if the string is empty and vice versa.

```
bool string::empty(){
```

```
    return length()==1;
}
```

d) Modifier:

- push_back(char a): Appends character a to the end of the string.

```
void string::push_back(char a)
{
    int length = strlen(str);
    char* tmp = new char[length + 2];
    for (int i = 0; i < length; i++) {
        tmp[i] = str[i];
    }
    tmp[length] = a;
    tmp[length + 1] = '\0';
    *this = string{ tmp };
    delete[] tmp;
}
```

- pop_back(): Removes the last character of the string

```
void string::pop_back()
{
    int length = strlen(str);
    char* tmp = new char[length];
    for (int i = 0; i < length - 1; i++)
        tmp[i] = str[i];
    tmp[length-1] = '\0';
    *this = Mystring{ tmp };
    delete[] tmp;
}
```

- swap(string& other): The content of the container is swapped with the content of str, which is a string object.

```
void string::swap(string& other)
{
    string tmp{ other };
    other = *this;
    *this = tmp;
}
```

e) String operations:

- copy(char s[], int len, int pos): Copies a substring from the string object's current value into the array indicated by s. The len characters that begin at position pos are contained in this substring.

```
void string::copy(char s[], int len, int pos)
{
   for (int i = 0; i < len; i++) {
      s[i] = str[pos + i];
   }
   s[len] = '\0';
}
```

f) Non-member function overloads:

- operator<<(ostream& os, const string& obj): insert the string into stream.

```
ostream& operator<<(ostream& os, const string& obj)
{
   os << obj.str;
   return os;
}
```

- istream& operator>>(istream& is, string& obj): extract the string from stream.

```
istream& operator>>(istream& is, string& obj)
{
   char* tmp = new char[100000];
   memset(&tmp[0], 0, sizeof(tmp));
   is >> tmp;
   obj = string{ tmp };
   delete[] tmp;
   return is;
}
```

II. Vector class:

1. Definition:

The vector class in the C++ Standard Library is a class template for sequence containers. A vector is a linear array of items of a specified type that permits quick random access to any entry. When random-access efficiency is critical, a vector is the optimal container for a sequence.

Syntax:

```
template <class Type, class Allocator = allocator<Type>>
class vector
```

Define:

```
template <typename T> class vector
{
```

```
    T* array;
    int capacity;
    int length;

public:
// etc
}
```

2. Properties:

- Sequence : Sequence containers have elements that are organized in a precise linear manner. The location of individual items in this sequence is used to access them.

- Dynamic array: Allows direct access to each element in the series, even using pointer arithmetics, and allows for reasonably quick addition and removal of items at the end.

- Allocator-aware : To dynamically address its storage demands, the container employs an allocator object.

3. Public member function:

a) Constructor:

- Default: vector() - Creates a container that is empty and has no elements.

```
vector::vector()
    {
        array = new T[1];
        capacity = 1;
        length = 0;
    }
```

- Fill: vector(int n, T val) - Constructs a container with n elements filled by val.

```
vector::vector(int n, T val)
{
    array = new T[n];
    for (int i=0;i<n;i++)
    array[i]=val;
    capacity = n;
    length = n;
}
```

- Copy: vector(const vector& other) - Constructs a container with n elements filled by val.

```
vector::vector(const vector& other)
{
    array = new T[other.capacity];
    capacity = other.capacity;
    length = other.length;
```

```
    for (int i=0;i<length;i++)
        array[i]=other.array[i];
}
```

b) Destructor:

- This frees up all of the storage space allocated by the vector's allocator.

```
vector::~vector()
{
    delete []array;
    array=nullptr;
    capacity = 0;
    length = 0;
}
```

c) Capacity:

- size(): return size of vector

```
    int vector::size()
    {
        return length;
    }
```

- emty(): return true if vector is empty and vice versa

```
    bool vector::empty(){
        return length==0;
    }
```

- capacity(): size of allocated storage capacity

```
    int vector::capacity()
    {
        return capacity;
    }
```

d) Element access:

- operator[]: Access element operator

```
T& vector::operator[](int index)
{
    if (index >= length) {
        exit(0);
    }
    return array[index];
```

```
}
```

- front(): access first element

```
T& vector::front()
{
    return array[0];
}
```

- back(): access last element

```
T& vector::back()
{
    return array[length-1];
}
```

e) Modifier:

- assign(T value): assign new value to the vector by replacing the old

```
void vector::assign(T value)
{
    for (int i=0;i<length;i++)
    array[i]=value;
}
```

- push_back(T data): Append the element data

```
void vector::push_back(T data)
{
    if (length == capacity) {
        T* temp = new T[2 * capacity];
        for (int i = 0; i < capacity; i++) {
            temp[i] = array[i];
        }
        delete[] array;
        capacity *= 2;
        array = temp;
    }
    array[length] = data;
    length++;
}
```

- pop_back(): delete last element

```
void vector::pop_back() { length--; }
```

- insert(int index, T data): insert element data at index

```
void vector::insert(int index, T data)
{
    if (index == capacity)
        push(data);
    else
        array[index] = data;
}
```

- erase(int pos): erase element at index pos

```
void vector::erase(int pos)
{
    for (int i=pos;i<length-1;i++)
    array[i]=array[i+1];
    length--;
}
```

- clear(): clear vector content

```
void vector::clear()
{
    length=0;
}
```