CPT-281 - Introduction to Data Structures with C++

Module 2

Vectors

Dayu Wang

- **What are the limitations of a regular array?**

  → **Size of the array cannot be changed once the array is initialized.**

```
const unsigned int SIZE = 10;
int arr[SIZE];
// Size of "arr" will be 10 "forever".
```

  It is not convenient to insert more items to a regular array.

  It is not convenient to delete items from a regular array.

- **Design of Vector**

  → **Size of the vector can dynamically change.**

  Size of the vector always represents the number of items in the vector.

  → **Fast and conveniently adding/removing items to/from vectors.**

  → **Fast accessing items stored in vectors (index-based).**

  → **The data structure should be applicable to all data types, even for user-defined classes (template class).**

- **Implementations**

  → **C++ STL:** `vector` (`#include <vector>`)

  → **Java:** `ArrayList` (`import java.util.ArrayList;`)

- **Vector Implementation**

  → **We will implement our own `Vector` class.**

- **In C++, vectors are implemented using pointers.**

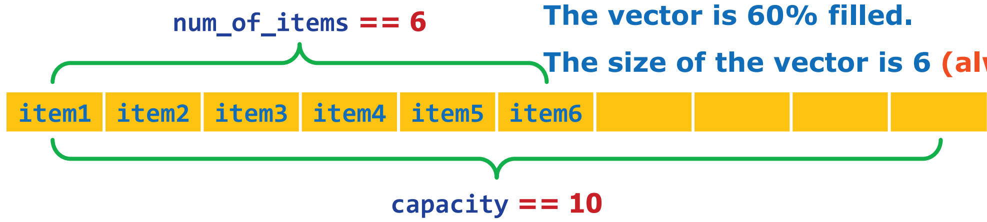
  → **Private data fields**

```
private:
    T* arr;  // Array -> Data container
    size_t capacity;  // Size of {arr}
    size_t num_of_items;  // Number of items stored in the vector
    static const size_t DEFAULT_CAPACITY;  // Used as capacity if no other specifications
```

What is the difference between `num_of_items` and `capacity`?

The vector is 60% filled.

The size of the vector is 6 (always the same as `num_of_items`).



The `DEFAULT_CAPACITY` is set to 10.

```
template<class T>
const size_t Vector<T>::DEFAULT_CAPACITY = 10;
```

→ **Default constructor**

```
template<class T>
Vector<T>::Vector() : capacity(DEFAULT_CAPACITY), num_of_items(0) {
    arr = new T[capacity];
}
```

- **C++ does not initialize variables for the user.**
- **In each constructor, you must initialize all the variables defined in the class.**

→ **If pointers are in the class, we must overload the "big three":**

1) **Assignment operator**
2) **Copy constructor**
3) **Destructor**

- **Shallow Copy and Deep Copy**

→ **A pointer points to a chunk of memory.**

Pointer `p` points to a chunk of memory.

`p`'s value is the memory address it points to.

If we use "`q = p;`" to make a copy of the pointer...

`q`'s value will be the same as `p`'s value.

`q`'s value will be the address of the same memory chunk `p` is pointing to.

`q` points to the same chunk of memory.

This is called **shallow copy**.

If we execute "`delete p;`" then the memory chunk is deleted.

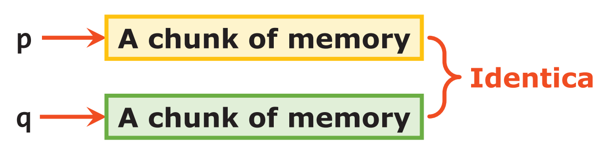
But `q` **never** noticed that the memory chunk it points to is **no longer** exist.

→ **Advantage of shallow copy: fast (only memory addresses are copied).**

→ **Disadvantage of shallow copy: "dangerous" (unreliable)**

- **Shallow Copy and Deep Copy**

→ **A pointer points to a chunk of memory.**



Pointer `p` points to a chunk of memory.

`p`'s value is the memory address it points to.

This time, we allocate a new chunk of memory (`new`).

Copy the content of `p`'s memory to the new memory chunk.

Let `q` point to the new memory chunk.

Although the content of `p` and `q` are identical, they are pointing to different memory addresses.

If we change or delete either one, it will **not** ~~affect the other~~.

This is called **deep copy**.

→ **Advantage of deep copy: "safe" (reliable)**

→ **Disadvantage of deep copy: slow (actual data are copied)**

→ Assignment Operator

Step 1: Avoid self-assignment.

Step 2: Delete dynamically allocated memory.

Step 3: Copy static data.

Step 4: Copy dynamic data.

Step 5: Return.

```
template<class T>
const Vector<T>& Vector<T>::operator = (const Vector<T>& rhs) {
    if (this != &rhs) {
        if (arr) {
            delete[] arr;
            arr = NULL;
        }
        capacity = rhs.capacity;
        num_of_items = rhs.num_of_items;
        if (capacity) {
            arr = new T[capacity];
            for (size_t i = 0; i < num_of_items; i++) { arr[i] = rhs.arr[i]; }
        }
    }
    return *this;
}
```


→ Copy Constructor

```
template<class T>
Vector<T>::Vector(const Vector<T>& other) {
    arr = NULL;
    *this = other;
}
```

→ Destructor

```
template<class T>
Vector<T>::~Vector() {
    if (arr) {
        delete[] arr;
    }
}
```

→ Other Constructors

```
Vector(size_t);  // Constructs a vector with initial size.
Vector(size_t, const T&);  // Fills each cell in the vector with a specific value.
```

See sample code for the implementations.

→ Subscript Operator

The subscript operator ([]) needs to be overloaded <u>twice</u> in the `Vector` class.

1) Used on the left side of assignment.

[Example] `vec[2] = 5;`

It is <u>modifiable</u>, so the operator function <u>returns a reference</u>.

2) Used on the right side of assignment.

[Example] `vec[6] = vec[2] + 1;`

It is unmodifiable, so the operator function <u>returns a const reference</u>.

```
T& operator [] (int);  // lvalue
const T& operator [] (int) const;  // rvalue
```

```
template<class T>
T& Vector<T>::operator [] (int index) {
    return arr[index];
}  // Time complexity: O(1)

template<class T>
const T& Vector<T>::operator [] (int index) const {
    return arr[index];
}  // Time complexity: O(1)
```


→ What's the difference between "at()" and "[]"?

`at()` **checks boundaries for the index, and <u>throws an exception</u> if "index out of bounds".**

`[]` **does not check boundaries for the index, and the program crashes if "index out of bounds".**

```
template<class T>
T& Vector<T>::at(int index) {
    if (index < 0 || index >= num_of_items) {
        throw std::out_of_range("Index out of bounds: " + std::to_string(index));
    }
    return arr[index];
}  // Time complexity: O(1)

template<class T>
const T& Vector<T>::at(int index) const {
    if (index < 0 || index >= num_of_items) {
        throw std::out_of_range("Index out of bounds: " + std::to_string(index));
    }
    return arr[index];
}  // Time complexity: O(1)
```

It is recommended to use "at()", not "[]".

→ **size() function**

The function returns the number of items stored in the vector.

```
template<class T>
size_t Vector<T>::size() const {
    return num_of_items;
}  // Time complexity: O(1)
```

→ **empty() function**

The function returns true if the vector is empty; false otherwise.

```
template<class T>
bool Vector<T>::empty() const {
    return !size();  // This is equivalent to "return size() == 0;".
}  // Time complexity: O(1)
```


→ **front() and back() functions**

```
template<class T>
T& Vector<T>::front() {
    if (empty()) {
        throw std::exception("Attempt to
access item in empty vector.");
    }
    return arr[0];
}  // Time complexity: O(1)

template<class T>
const T& Vector<T>::front() const {
    if (empty()) {
        throw std::exception("Attempt to
access item in empty vector.");
    }
    return arr[0];
}  // Time complexity: O(1)
```

```
template<class T>
T& Vector<T>::back() {
    if (empty()) {
        throw std::exception("Attempt to
access item in empty vector.");
    }
    return arr[size() - 1];
}  // Time complexity: O(1)

template<class T>
const T& Vector<T>::back() const {
    if (empty()) {
        throw std::exception("Attempt to
access item in empty vector.");
    }
    return arr[size() - 1];
}  // Time complexity: O(1)
```

→ **The resize operation**

Before resize

`capacity == 5, num_of_items == 5`

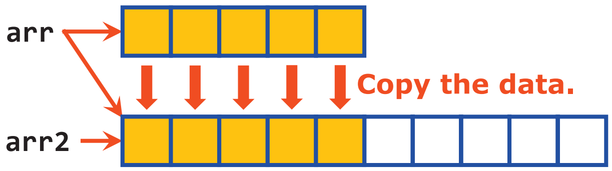
The vector is full-filled.

After resize

`capacity == 10, num_of_items == 5`

→ **How to change the capacity?**

A dynamic array's (pointer's) size **cannot** be directly changed. We have to "work around" it.



`capacity == 5` `num_of_items == 5`

Double the capacity.

`capacity == 10`

```
template<class T>
void Vector<T>::resize() {
    T* arr2 = new T[capacity *= 2];
    for (size_t i = 0; i < num_of_items; i++) { arr2[i] = arr[i]; }
    delete[] arr;
    arr = arr2;
}
```


→ **Why each time when we resize, we double the capacity?**

Why **not** triple the capacity?

Why **not** add 100 to the capacity?

→ **Each resize operation is $O(n)$.**

→ **Spread the cost of copying.**

Number of resize operations	Size of array
1	$10 \times 2^{1} = 20$
2	$10 \times 2^{2} = 40$
3	$10 \times 2^{3} = 80$
4	$10 \times 2^{4} = 160$
10	$10 \times 2^{10} = 10240$
15	$10 \times 2^{15} = 327680$
20	$10 \times 2^{20} = 10485760$

- We do the resize after adding n items, so on average, we need a copy for each item.
- Therefore, effective resize operation is $O(1)$.

→ **push_back() function**

push_back() function adds an item to the end of the vector.

```
template<class T>
void Vector<T>::push_back(const T& item) {
    if (capacity == num_of_items) { resize(); }
    arr[num_of_items++] = item;
}  // Time complexity: O(1)
```

→ **How about inserting items to the front or middle of the vector?**

This needs to use iterators. We have not yet learned iterators.

We will come back talking about insertion in vectors after we learned iterators.

→ **pop_back() function**

pop_back() function removes the last item from the vector.

```
template<class T>
void Vector<T>::pop_back() {
    if (empty()) {
        throw std::exception("Accessing empty vector");
    }
    num_of_items--;
}  // Time complexity: O(1)
```


After we learned iterators, we will come back talking about the erase operation, which removes the item at specific position from the vector.

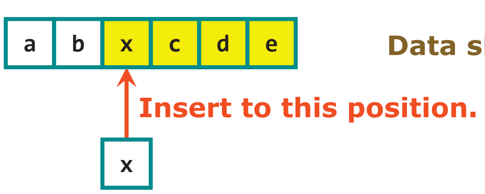
→ **Time complexity**

push_back(): O(1)

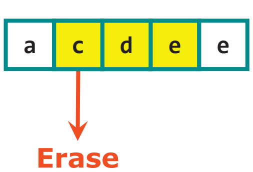
pop_back(): O(1)

insert(): $O(n)$

erase(): $O(n)$



Data shift is necessary.



Data shift is necessary.

→ **Overloading the stream insertion operator**

Stream insertion operator: "<<"

Stream extraction operator: ">>"

```cpp
template<class E>
std::ostream& operator << (std::ostream& out, const Vector<E>& vec) {
    out << '[';
    for (size_t i = 0; i < vec.size(); i++) {
        out << vec.at(i);
        if (i != vec.size() - 1) { out << ", "; }
    }
    out << ']';
    return out;
}  // Time complexity: O(n)
```

```cpp
Vector<int> vec(5, 1);
for (size_t i = 1; i < vec.size(); i++) {
    vec.at(i) = vec.at(i - 1) + 1;
}
cout << vec << endl;
```

Console `[1, 2, 3, 4, 5]`

- **<u>What are the advantages of vectors?</u>**

→ **Fast access (index based):** O(1)

→ **Dynamically resized by the resize operation.**

Adding item to the end: O(1)

→ **Index-based structure can be conveniently iterated through.**

- **<u>What are the disadvantages of vectors?</u>**

→ **Slow insertion/erase:** O(n)

Data shift (done by loop) is inevitable.

→ **Unused space exists.**

In most cases, the `num_of_items` **is <u>smaller than</u>** `capacity`**.**

- **<u>More topics about vectors</u>**

→ **Iterators**

→ "`insert()`" **and** "`erase()`" **functions**

→ **Search functions**

Search functions need to use iterators, and are not inside the vector class.