DATA STRUCTURES AND ALGORITHMS

Lecture 04

Learning outcome 2

Abstract and concrete data types

- Abstract data type (ADT) represents the user's desire for functionalities
  - Defines the data type in terms of supported operations and their complexity
  - It says nothing about how it will be implemented
  - Describes the data type from the point of view of the data type user
  - For example: a list is an ADT that allows you to insert a value at its end in $O(1)$
- Concrete data type is an implementation of an abstract type
  - For example: `vector<T>` is a list implementation in C ++
    - In C# it is `List<T>`, in Java it is `ArrayList`, ...

LIST

Operations on the list

- The ADT list is nowhere formally defined
  - „The ADT List is a linear sequence of an arbitrary number of items” (source: doc.ic.ac.uk)
- List of possible operations on the list:
  - Making an empty list
  - Insert a new item at a position in the list
  - Remove an item from a position in the list
  - Check if the list is empty or not
  - Retrieve an item at a position in the list
  - Retrieve the number of items in the list

Other possible operations

- Lets come up with three more operations that make sense on the list
  - Insert an element at the end
  - Retrieve the first element
  - Retrieve the last element
  - Remove all items from the list
  - Search for the first occurrence of the value in the list
  - Search for all occurrences of the value in the list
  - Retrieve the next element from a position
  - Retrieve the previous element from a position
  - ...

VECTOR

Vector as a concrete ADT list

- The vector is a C++ implementation of the ADT list
  - It contains a number of methods that represent operations on a list, for example:
    - We can make an empty list (constructor)
    - We can insert a new element at a position in the list (`insert` method)
    - We can remove an element from a position in the list (`remove` method)
    - ...
- Unlike the ADT list, the vector is concrete and ready to use
  - Implemented as a generic class `vector<T>`

ALGEBRA

How vector works

- A vector is a wrapper around a dynamic array
  - The elements of the vector are placed one behind the other in memory
  - We can access them by using pointers
- Resizing the array is done automatically, as needed
  - When the array is filled, the next (expensive) operation occurs:
    - A new, larger dynamic array is allocated
    - All elements from the old array are copied to the new array
    - The old field is deallocated
  - Optimization: vector growth usually occurs exponentially
    - Objective: to avoid growth with each insertion and to provide an $O(1)$ for insertion at the end

ALGEBRA

Vector creation and destruction (1/2)

- There are six basic ways to make a vector:
  - `vector<int> one;`
    - Creates an empty vector (*default*)
  - `vector<int> two(n);`
    - Creates a vector of *n* elements initialized to the default value (*fill*)
  - `vector<int> three(n, val);`
    - Creates a vector of *n* elements, each a copy of a `val` (*fill*)
  - `vector<int> four(three.begin(), three.end());`
    - Creates a vector by copying elements from a given range (*range*)
      - The first value is the start address (the element at that address is also taken)
      - The second value is the last address (the item at that address is not taken)

ALGEBRA

Vector creation and destruction (2/2)

- `vector<int> five(three);`
  - Creates a vector by copying all elements from a given vector (*copy*)
- `vector<int> six({ 11, 22, 33 });`
  - Creates a vector by copying all elements from the initialization list (*initializer list*)

- The vector is automatically destroyed at the end of the function in which it is declared
  - If a vector stores objects, a destructor is called on each

ALGEBRA

Copying a vector

- `operator=` copies elements from the right vector to the left
  - The previous content of the left vector is destroyed
  - The size of the left vector may change after copying
  - For copying to be possible, both vectors must be of the same data type T
- Example:

```
vector<int> one(3, 404);
vector<int> two(5, 701);

two = one;

for (unsigned i = 0; i < two.size(); i++) {
      cout << two[i] << endl;
}
```


ALGEBRA

Vector size and capacity

- For the vector v we distinguish two measures:
  - `v.size()` returns the number of elements placed in the vector by the user
  - `v.capacity()` returns the size of the allocated dynamic array (also expressed in number of elements)
  - When `size()` should become greater than `capacity()`, vector will grow
- Example:

```
vector<int> one;
for (unsigned i = 0; i < 100; i++) {
    cout << "size=" << one.size() << " (capacity=" <<
        one.capacity() << ")" << endl;
    one.push_back(i);
}
```


ALGEBRA

Manual change of vector size and capacity

- We can also explicitly change the size and capacity:
  - `v.resize(n);`
    - Changes the size of the vector to exactly *n* elements
    - If *n* is less than the current size, the end elements are discarded
      - Capacity does not change
    - If *n* is larger than the current size, elements are added at the end
      - If *n* is greater than the current capacity, the vector grows
  - `v.reserve(n);`
    - It changes the capacity of a vector so that it can contain at least *n* elements
      - If *n* is greater than the current capacity, the vector grows
      - If *n* is less than the current capacity, the method does nothing

Example

```
vector<int> v;
v.push_back(10);
v.push_back(20);
v.push_back(30);

v.resize(0);
cout << "s=" << v.size() << ", c=" << v.capacity() << endl;

v.resize(38);
cout << "s=" << v.size() << ", c=" << v.capacity() << endl;

v.reserve(100);
cout << "s=" << v.size() << ", c=" << v.capacity() << endl;

v.reserve(75);
cout << "s=" << v.size() << ", c=" << v.capacity() << endl;
```

Access to elements

- The vector offers several ways to access the elements:
  - `v[i]` returns a reference to an element on index *i*
    - If *i* is out of range, the behavior is not defined
  - `v.at(i)` also returns a reference to an element on index *i*
    - If *i* is out of range, throws an exception of type `out_of_range`
  - `v.front()` returns a reference to the first element
    - If the vector is empty, the behavior is not defined
  - `v.back()` returns a reference to the last element
    - If the vector is empty, the behavior is not defined

Example

```
vector<int> one(5);

for (unsigned i = 0; i < one.size(); i++) {
      one[i] = (i + 1) * 10;
}

for (unsigned i = 0; i < one.size(); i++) {
      cout << one.at(i) << " ";
}
cout << endl;

cout << one.front() << " " << one.back() << endl;
```

Vector modifiers (1/4)

- Vector modifiers change its content:
  - `v.assign()` is similar to constructors *fill*, *range* and *initializer list*

```
v.assign(7, 100);
v.assign(x.begin(), x.end());
v.assign({ 11, 22, 33 });
```

    - All elements previously contained in the vector are destroyed
    - If the new size is larger than the current capacity, the vector will grow
  - `v.push_back(val)`
    - Adds a copy of the `val` to the end of the vector
    - May cause vector growth
    - Preferred way to fill the vector because it does not require moving the other elements

Vector modifiers (2/4)

  - `v.pop_back()`
    - Removes and destroys the last element (and reduces the size by 1)
    - Preferred removal method
  - `v.insert()` inserts one or more elements into a given position:
    - The first parameter is the position, the others are similar to the constructors:

```
v.insert(v.begin() + 3, 99);
v.insert(v.begin() + 3, 10, 99);
v.insert(v.begin() + 3, v.begin(), v.end());
v.insert(v.begin() + 3, { 11, 22, 33 });
```

    - May cause vector growth
    - If the insertion is not done at the end of the vector, all other elements behind the new ones will be moved to the right – bad for performance

Vector modifiers (3/4)

- `v.erase()` removes one or more elements:

```
v.erase(v.begin() + 3);
v.erase(v.begin() + 3, v.end());
```

  - Removes and destroys an element at a given position or in a given range (and reduces the size by the number of removed elements)
  - Returns the iterator to the next element after the deleted ones
    - All existing iterators pointing to the indices behind the deleted ones become invalid
  - If the removal is not done from the end of the vector, all other elements behind the removed ones will be moved to the left – bad for performance

Vector modifiers (3/3)

- `v.clear()` completely empties the vector and destroys all elements
  - Size goes to 0, capacity may or may not change (depends on implementation)
  - In the case of objects, a destructor is called on each element

Example

```
vector<int> v(5, 0);

v.pop_back();
v.pop_back();

v.push_back(10);

v.insert(v.begin(), 2, 20);

v.erase(v.begin() + 2);
v.erase(v.begin() + 2);

for (unsigned i = 0; i < v.size(); i++) {
	cout << v[i] << " ";
}
cout << endl;
```


Other important methods

- `v.empty()` returns whether the vector is empty or not

ITERATORS AND IN-PLACE OBJECT CONSTRUCTION

Iterators

- An iterator is a standard way to access data contained in a container (vector, map, list,...)
- An iterator is any object that has the characteristics:
  - Allows access to an element (`*it`)
  - Allows moving to the next element (`++it`)
  - Optionally, allows moving to the previous element (`--it`)
- The pointer is also an iterator because it satisfies the above conditions
- Some containers also have alternative ways to access the elements, but iterators are universal for all containers
  - For example, `[]` or `at()`

Vector iterators

- The vector contains several types of iterators:
  - `vector<T>::iterator` is a class whose ++ leads towards the end
  - `vector<T>::reverse_iterator` is a class whose ++ leads towards the beginning
- Methods that return iterators:
  - `v.begin()` – returns the iterator to the first element
  - `v.end()` – returns the iterator to the first element after the end
  - `v.rbegin()` – returns the iterator to the reverse start (which is the last element)
  - `v.rend()` – returns the iterator to the reverse end (which is the element directly in front of the first element in the vector)

Example

```
vector<int> v;

for (int i = 10; i <= 50; i += 10) {
    v.push_back(i);
}

for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    cout << *it << endl;
}

for (vector<int>::reverse_iterator it = v.rbegin();
                                          it != v.rend(); ++it) {
    cout << *it << endl;
}
```

Removing from vector (1/2)

- How to remove all even numbers from the vector?

```
vector<int> v({ 11, 22, 33, 44, 55 });
```

- When removing from the vector, we must consider:
  - „erase … invalidates iterators and references at or after the point of the erase …"
  - Therefore, this removal method is incorrect :

```
for (auto it = v.begin(); it != v.end(); ++it) {
    if (*it % 2 == 0) {
        v.erase(it);
    }
}
```

  - Reason: after the first deletion the iterator `it` is no longer valid and we must not increase it with ++

Removing from vector (2/2)

- Two main ways to remove:
  - STL function `remove_if`

```
auto x = remove_if(v.begin(), v.end(), should_i_delete);
v.erase(x, v.end());
```

  - Minor modification of incorrect deletion to make it correct:

```
for (auto it = v.begin(); it != v.end(); ) {
    if (*it % 2 == 0) {
        it = v.erase(it);
    }
    else {
        ++it;
    }
}
```

Is there a difference between ++it and it++?

- The following example demonstrates the work of prefix and postfix operators (and there is nothing new for us):

```
int x = 10;
cout << x++ << endl;
cout << ++x << endl;
```

- Is there a difference between the next two loops?

```
for (auto it = v.begin(); it != v.end(); it++) {
    cout << *it << endl;
}

for (auto it = v.begin(); it != v.end(); ++it) {
    cout << *it << endl;
}
```

- There is no difference in the result - both display the same
  - However, differences in performance may exist

What is the difference between ++it and it++?

- The prefix on the object works like this:
  - Change the original object
  - Returns the reference to the original object
- The postfix on the object works like this :
  - Create a new temporary object by copying the original
  - Change the original object
  - Return the copy
- Postfix uses additional copying, which is usually bad
  - There is no difference in the built-in data types
  - There is a chance that the optimizer will avoid copying
  - If we use a prefix, we can't go wrong!

Unnecessary copying of objects

- Lets take a look at this structure:

```
struct Rectangle {
      Rectangle(int s, int v) {
            this->width = s;
            this->height = v;
     }
     int width;
     int height;
};
```

- How can we add a new rectangle to the end of the vector?

```
vector<Rectangle> vp;
Rectangle p(17, 4);
vp.push_back(p);
```

- How many objects have we created and can we solve the problem smarter?

Methods emplace() and emplace_back()

- Method `emplace()` behaves just like `insert()`, but instead of copying it constructs an object at the target location
  - `emplace_back()` constructs the object at the end
- Both methods receive a variable number of parameters:
  - The first parameter is always the position (just for `emplace()`)
  - The other parameters are in fact the values that go to the appropriate constructor
- The solution to our problem from the last slide:

```
vector<Rectangle> vp;
vp.emplace_back(17, 4);
```

The complexity of some operations

Method	Complexity
vector<T> v;	O(1)
vector<T> v(n);	O(n)
vector<T> v(n, value);	O(n)
vector<T> v(begin, end);	O(n)
v[i];	O(1)
v.at(i);	O(1)
v.size();	O(1)
v.empty();	O(1)
v.begin();	O(1)
v.end();	O(1)
v.front();	O(1)
v.back();	O(1)
v.capacity();	O(1)

Method	Complexity
v.push_back(value);	O(1)
v.insert(iterator, value);	O(n)
v.pop_back();	O(1)
v.erase(iterator);	O(n)
v.erase(begin, end);	O(n)

Problem

1. Implement your simple vector for storing integers. Let the strategy of increasing the capacity be that the new capacity is always 50% bigger than the previous one. Define the following operations on the vector:
   a) Creating a vector and initializing it with a list
   b) Getting size and capacity
   c) Inserting an element at the end
   d) Retrieving an element at position *i*

Source.cpp

```
MyVector mv({ 11, 22, 33, 44, 55 });
mv.push_back(66);
mv.push_back(77);
mv.push_back(88);
mv.push_back(99);

cout << "s=" << mv.size() << ", c=" << mv.capacity() << endl;
for (int i = 0; i < mv.size(); ++i) {
    cout << mv.at(i) << endl;
}
```


MyVector.h

```
#pragma once
#include <initializer_list>

class MyVector {
private:
    int* numbers;
    int s;
    int c;
    void grow();

public:
    MyVector(std::initializer_list<int> il);
    ~MyVector();
    int size();
    int capacity();
    void push_back(int value);
    int at(int i);
};
```

MyVector.cpp (1/3)

```
#include "MyVector.h"

MyVector::MyVector(std::initializer_list<int> il) {
    numbers = new int[il.size()];
    int i = 0;
    for (auto it = il.begin(); it != il.end(); ++it) {
        numbers[i++] = *it;
    }
    s = il.size();
    c = il.size();
}

MyVector::~MyVector() {
    delete[] numbers;
}
```


MyVector.cpp (2/3)

```
int MyVector::size() {
    return s;
}

int MyVector::capacity() {
    return c;
}

void MyVector::push_back(int value) {
    if (c == s) {
        grow();
    }
    numbers[s++] = value;
}

int MyVector::at(int i) {
    return numbers[i];
}
```

MyVector.cpp (3/3)

```
void MyVector::grow() {
    // Allocate new array
    c = c * 1.5;
    int* novi = new int[c];

    // Copy values old => new
    for (int i = 0; i < s; i++) {
        novi[i] = numbers[i];
    }

    // Deallocate old.
    delete[] numbers;

    // Copy address of the new array
    numbers = novi;
}
```