

# C++

## Containers

1

# Introduction

- A container is an object that stores other objects (its elements) and that has methods to access its elements
- There are three important concepts:
  - The container type (sequence, associative container, container adaptors)
  - The type of objects contained (int, double, user defined class)
  - The operations on the containers (and their complexity)

2

# Example

- We will illustrate how containers works (and can be implemented) by an example
- We will implement an array: i.e a sequence of homogeneous element
- Operations:
  - Assignment
  - Assignment and access to a random element

3

# Example: Array

```
#ifndef ARRAY_H
#define ARRAY_H

template <class T>
Array {
private:
    int _size;
    T* _data;
public:
    Array(int sz=100);
    Array(const Array<T>& a);
    ~Array();

    int size() const { return _size; }
    Array<T>& operator= (const Array<T>& a);
    T& operator[] (int i) { return _data[i]; }
    const T& operator[] (int i) const { return _data[i]; }
};

#endif
```

Default ctor

Copy ctor

Assignment operator

Access to an element

4

## Ctor and Dtor

```
template<class T>
Array<T>::Array(int sz) : _size(sz), _data(new T[sz]){}

template<class T>
Array<T>::Array(const Array<T>& a) : _size(a._size), _data(new T[a._size]) {
    for (int i=0; i < _size; i++) {
        _data[i] = a._data[i];
    }
}

template<class T>
Array<T>::~~Array() {
    delete[] _data;
}
```

5

## Copy Ctor: reminder

- Remember: if a class owns data (has member data which are pointer), then you should provide a copy constructor
- The default copy constructor does only bit-wise copy
- Example:  
    Array<int> A;  
    Array<int> B = A; // or B(A)
- With the default copy ctor, \_data in B and A will point to the same location
- Deleting \_data, for example in B dtor, will also delete the memory allocated for A.\_data and potentially create memory bugs

6

## Assignment operator

- Let us look first at a wrong implementation of the assignment operator, we will then comment and fix it

```
template<class T>
Array<T>&
Array<T>::operator= (const Array<T>& a) {
    delete[] _data;
    _size = a._size;
    _data = new T[_size];
    for (int i =0; i < _size; i++){
        _data[i] = a._data[i];
    }
    return *this;
}
```

The problem is here

7

## Assignment operator problem

- Suppose that we write:  
    Array<int> a;  
    a = a; // problem!
- Remember the lesson on assignment operator: we need to be careful about self-assignment

8

## Assignment operator fixed

```
template<class T>
Array<T>&
Array<T>::operator= (const Array<T>& a) {
    if (this != &a) {
        delete[] _data;
        _size = a._size;
        _data = new T[_size];
        for (int i = 0; i < _size; i++){
            _data[i] = a._data[i];
        }
    }
    return *this;
}
```

We can now write:  
Array<int> a;  
a = a;

9

## Printing facilities

- Let us add some printing facilities for the class array
- We want to do:  
Array<int> a;  
cout << a << endl;

```
template <class T>
ostream&
operator<< (ostream& os, const Array<T>& a){
    for (int i = 0; i < a.size(); i++) {
        os << a[i] << " ";
    }
    return os;
}
```

Note: we already saw how to overload "<<" in the prev. lesson on op. overloading

10

## Usage

```
#include <iostream>
#include <Array.h>

using namespace std;

int main() {
    Array<int> a(20);
    for(int i = 0; i < a.size(); i++) {
        a[i] = i; // call operator[]
    }
    cout << a << endl;

    Array<int> b = a; // copy ctor
    cout << b.size() << endl;

    Array<int> c(50);
    c = a; // assignment op
}
```

11

## Remarks on operator[]

- There are two operator[] implementations in the class Array<T>:
  - template<class T> T& Array<T>::operator[] (int i);
  - template<class T> const T& Array<T>::operator[] (int i) const;
- Reason: suppose you want e.g. to pass a const Array to a function that will need to access some of the elements

12

## Remarks on operator[]

```
template <class T>
T f (const Array<T>& a)
{
    T sum = 0;
    for (int i = 0; i < a.size(); i++)
        sum += a[i];
    return sum;
}
```

a is const

Elements of a are read only but not modified

Without the second form of operator[], this code would not compile.

13

## Containers and the STL

- STL has 4 types of containers:
  - Sequences: list, vector, deque, ...
  - Associative containers: set, map, multiset, hash, ...
  - String
  - Container adaptors: stack, queue, priority\_queue

14

## Containers in the STL

- Sequences: variable sized container with elements arranged in a linear order; it supports insertion and removal of elements
- Available operations depends on the type of sequence: e.g. vector<T> allows random access by operator [], but not list<T>

15

## Containers in the STL

- Associative container: it is a container of pairs <key, value> objects
- For some assoc. containers key==value (sets) while for others they are different (maps)
- Some assoc. cont. require unique key (set, map) while other not (multiset, multimap)
- Operation availables depends on the assoc. container

16

# Container adaptors

- Container adaptor means that it is implemented on top of another container
- For example, the stack in the STL is defined as `Stack<T, Sequence>`
  - Sequence is the underlying container
  - By default deque is used
  - T is the type of the element
- Container adaptors provide a restricted subset of Container facilities