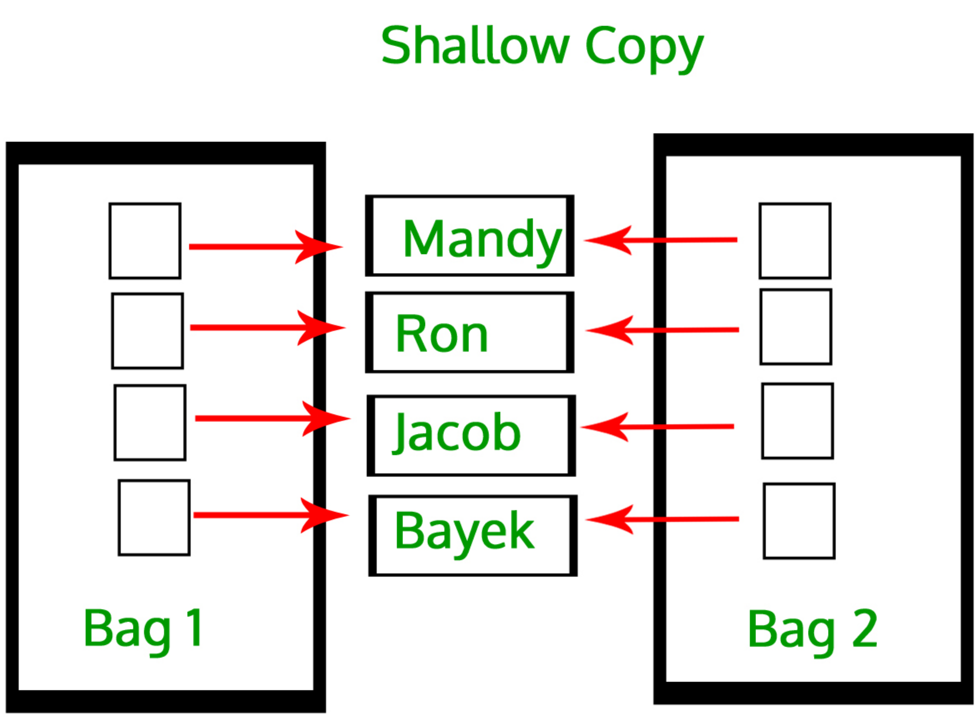
VE280 Final Review (L17-18)

Lecture 17: Deep Copy

Shallow Copy & Deep Copy

Motivation

C++ does **not know much about your class**, the ***default copy*** and ***default assignment operator*** it provides use a copying method known as a member-wise copy, also known as a *shallow copy*.



Example Code

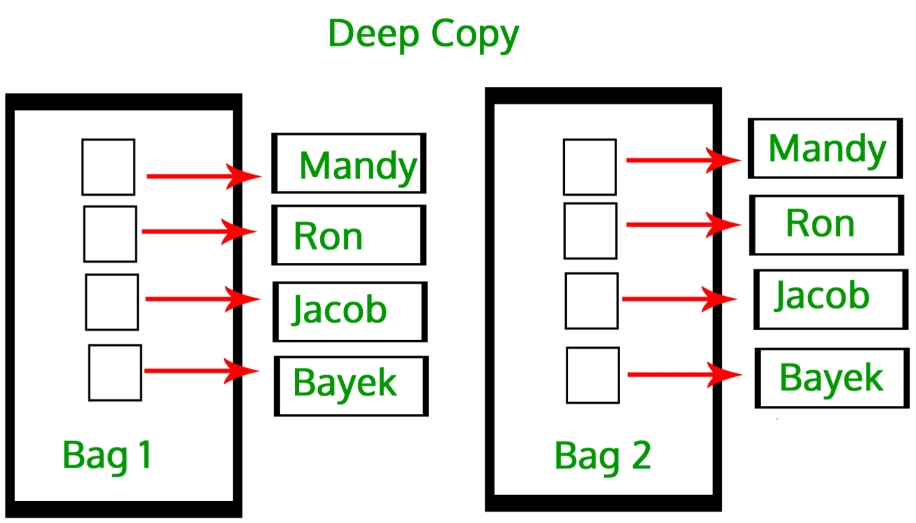
```
#include <iostream>
using namespace std;
const int MAX_CAPACITY = 10;
class Bag {
    string *items;
    public:
    Bag();
    void insert(string str); // implementation omitted
};
Bag::Bag() : items(new string[MAX_CAPACITY]) {}
int main() {
    Bag bag1;
    bag1.insert("VE280");
    Bag bag2 = bag1;
    return 0;
}
```

What is the terrible result?

1. When you change the value of items in bag2, then the items in bag1 also changes.
2. What if you have a destructor for the class?

What does deep copy do?

Instead, a **deep copy** copies all fields, and makes copies of dynamically allocated memory pointed to by the fields.



The Rule of the Big 3/5

If you have any dynamically allocated storage in a class, you must follow this Rule of the Big X, where X = 3 traditionally and X = 5 after c++11 (see `std::move()`).

> Whenever an object owns resources, any resources, not just memory, it should implement 5 methods: A constructor and a **destructor**, A **copy constructor**, a move constructor, a copy **assignment operator**, and a move assignment operator.

Structure

```
class MyClass {
    // Member variables
public:
    MyClass(MyClass &that); // Copy constructor
    MyClass &operator=(const MyClass &that); // Overload '=', assignment
operator
    void detroy(); // Destruct behaviour
    ~MyClass(){detroy();} // Destructor
    // Other member functions omitted
};
```

The rule: Traditionally **constructor/destructor/copy assignment operator** forms a rule of 3.

If you need one of them, then you need all of them. You should never leave them unsaid whenever dynamic allocation is involved.

Implementation

Usually, we would need to implement some private helper functions `removeAll()` and `copyFrom()`, and use them in the big 3. Consider a `Dlist` (in project 5).

- A destructor

```
template <class T>
Dlist<T>::~Dlist() {
    removeAll();
}
```

- A copy constructor

```
template <class T>
Dlist<T>::Dlist(const Dlist<T> &l): first(nullptr), last(nullptr) {
  copyAll(l);
}
```

- An assignment operator

```
template <class T>
Dlist<T> & Dlist::operator=(const Dlist<T> &l) {
    if (this != &l) {
        removeAll();
        copyAll(l);
    }
    return *this;
}
```

Exercise

Consider the following cases, which one/ones is/are called?

- ```
  Polynomial a(1,2);
  Polynomial b;
  b = a;
  ```

- ```
  Polynomial a(1,2);
  Polynomial b = a;
  ```

Lecture 18: Dynamic Resizing

Motivation

In many applications, we do not know ***the length of a list in advance***, and may need to grow the size of it when running the program. In this kind of situation, we may need dynamic resizing.

Array Example

When do we use Dynamic Resizing?

When the array is at maximum capacity, we will grow the array. Using `grow()` method:

- The `grow` method won't take any arguments or return any values.
- It should never be called from outside of the class, so add it as a **private** method taking no arguments and returning void.

How to implement a `grow()` function?

Four steps in general:

- **Make a new array** with desired size. For example,

  ```
  int *tmp = new int[new_size];
  ```

- **Copy** the elements from the original array to the new array iteratively. Suppose the original array is `arr` with size `size`.

  ```
  for (int i = 0; i < size; i++){
      tmp[i] = arr[i];
  }
  ```

- Replace the variable with the new array and **delete** the original array. Suppose the original array is `arr`:

  ```
  delete [] arr;
  arr = tmp;
  ```

- **Make sure all necessary parameters are updated**. For example, if the `size` of array is maintained, then we can do:

  ```
  size = new_size;
  ```

Common selections of `new_size`

- `size + 1`: This approach is simplest but most inefficient. Inserting `N` elements from capacity 1 needs `N(N-1)/2` number of copies.
- `2*size`: Much more efficient than `size+1`. The number of copies for inserting `N` elements becomes smaller than `2N`.
- What about even larger (eg: `size^2`)? Usually not good, for it occupies far too much memory.

Good Luck && Take Care! 👍

Reference

[1] Yunuo, Chen. VE280 FA2021 RC 7.

[2] Weikang, Qian. VE280 Lecture Slides 2022.