VE280 FA22 RC7

Lecture 17: Deep Copy

VE280 FA22 RC7

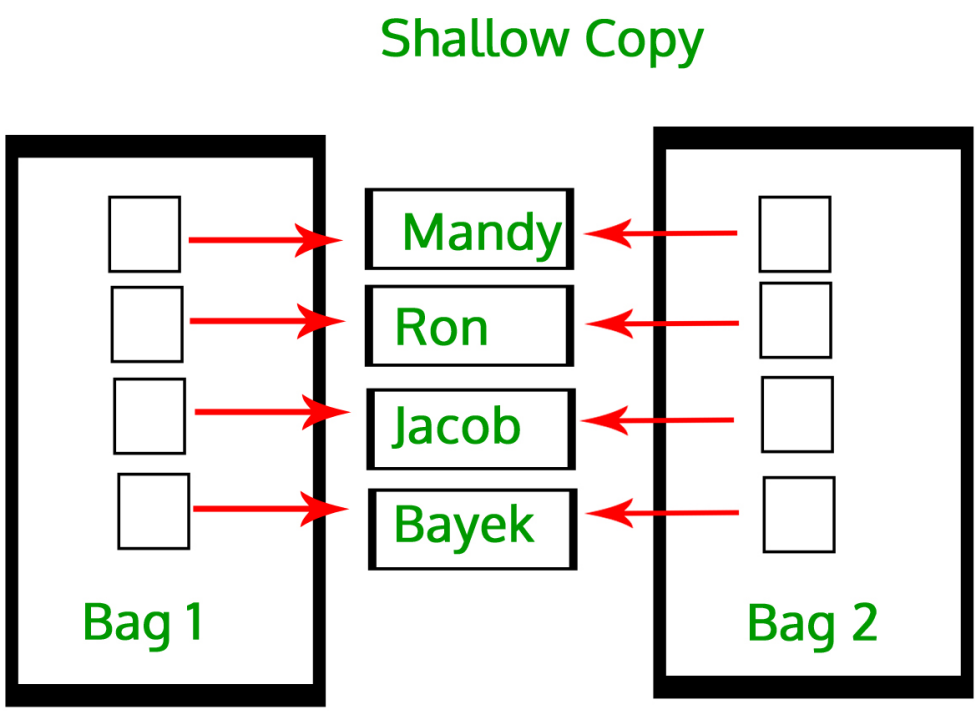
- Lecture 17: Deep Copy
  - Shallow Copy & Deep Copy
    - Motivation
    - Example Code
    - What is the terrible result?
    - What does deep copy do?
    - The Rule of the Big 3/5
      - Structure
      - `default` keyword
      - Implementation
    - Exercise
- Lecture 18: Dynamic Resizing
  - Why do we need Dynamic Resizing?
  - Array Example
    - When do we use Dynamic Resizing?
    - How to implement a `grow()` function?
    - Difference between delete and delete[]
  - Common selections of `new_size`
- Exercises
- Reference

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*.

This works well if the fields are ***values***, but may not be what you want for fields that point to ***dynamically allocated memory***. The pointer will be copied, **but the memory it points to will not be copied**: the field in both the original object and the copy will then point to the same dynamically allocated memory, this causes problem at erasure, causing `_____`.



Example Code

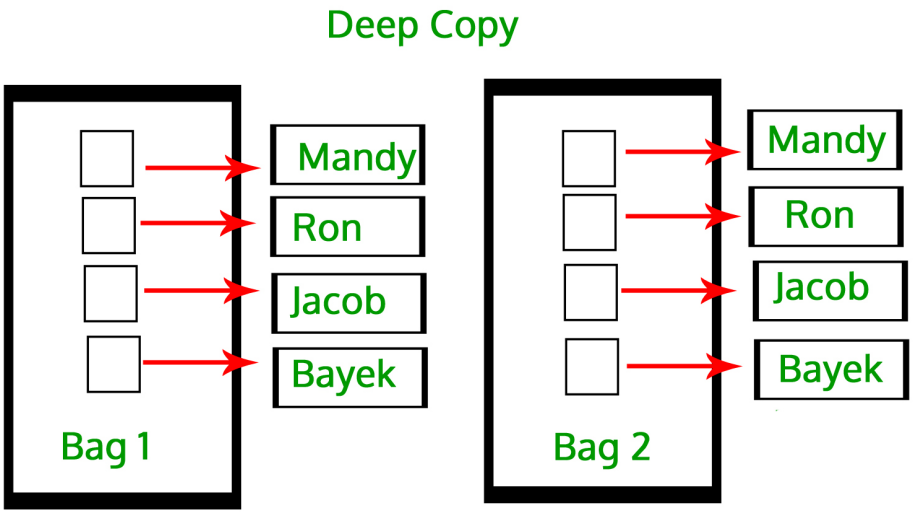
```
#include <iostream>
using namespace std;
const int MAX_CAPACITY = 10;
class Bag {
    string *items;
    public:
    Bag();
    void insert(string str); // implementation omitted
};
Bag::Bag() : items(new string[MAX_CAPACITY]) {}
int main() {
    Bag bag1;
    bag1.insert("VE280");
    Bag bag2 =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

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. Move semantics (`std::move()`) is a feature available after C++11, which is not in the scope of this course.

`default` **keyword**

If you want to use the version synthesized by the compiler, you can use `= default`:

```
Type(const Type& type) = default;
Type& operator=(Type&& type) = default;
```

Implementation

Usually, we would need to implement some private helper functions `removeAll()` and `copyFrom()`, and use them in the big 3. Consider a `Dlist` of `int` type as example.

- A destructor

```
Dlist::~Dlist() {
    removeAll();
}
```

- A copy constructor

```
Dlist::Dlist(const Dlist &l): first(nullptr), last(nullptr) {
  copyAll(l);
}
```

- An assignment operator

```
Dlist & Dlist::operator=(const Dlist &l) {
    if (this != &l) { // why ?
        removeAll();
        copyAll(l);
    }
    return *this;
}
```

Exercise

Recall binary tree and in-order traversal from Project 2. We define that a *good tree* to be a binary tree with **ascending in-order traversal**. Write the deep copy functions for the following codes.

```
class GoodTree {
        int *op;
        GoodTree *left;
        GoodTree *right;
public:
    void removeAll();
    // EFFECTS: remove all things of "this"
    void insert(int *op);
    // EFFECTS: insert op into "this" with the correct location
    //                  Assume no duplicate op.
};
```

You may use `removeAll` and `insert` in your `copyAll` method.

Your Answer here:

Lecture 18: Dynamic Resizing

Why do we need Dynamic Resizing?

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?

In general, there are four steps:

1. Allocate a bigger array.
2. Copy the smaller array to the bigger one.
3. Destroy the smaller array.
4. Modify elts/sizeElts to reflect the new array.

If the implementation of the list is a dynamically allocated array, we need the following steps to grow it:

- 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;
```

Difference between delete and delete[]

```
string *S = new string[3]; //They are PAIRED!!!!!
delete[] S;

string *s = new string;
delete s;
```

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.

Seems cost a lot to resize the array? But does it happen very often?

Learn more about amortized complexity in VE281/EECS281.

Exercises

1. To ensure a deep copy, what are the three methods that you should provide?
    1. `_____`
    2. `_____`
    3. `_____`
2. For each of the following codes, there might be some problems. Write down the problems and how to fix them? If there is none, write “None”.

1.
```
void study() {
    int * ptr = new int(280);
    int study1 = *ptr;
    ptr = new int(215);
    int study2 = *ptr;
    study2 += study1;
    delete ptr;
}
```

   Problem: _____

2.
```
class DoubleSet {
    // OVERVIEW: a mutable set of double numbers
    double *elts; // pointer to dynamic array
    int sizeElts; // capacity of the array
    int numElts; // current occupancy
public:
    DoubleSet &operator=(const DoubleSet &is);
    // other unrelated methods omitted.
};

DoubleSet &DoubleSet::operator=(const DoubleSet &is) {
    delete[] elts;
    sizeElts = is.sizeElts;
    elts = new double[sizeElts];
    for (int i = 0; i < is.sizeElts; i++)
        elts[i] = is.elts[i];
    return *this;
}
```

   Problem: _____

Reference

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

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