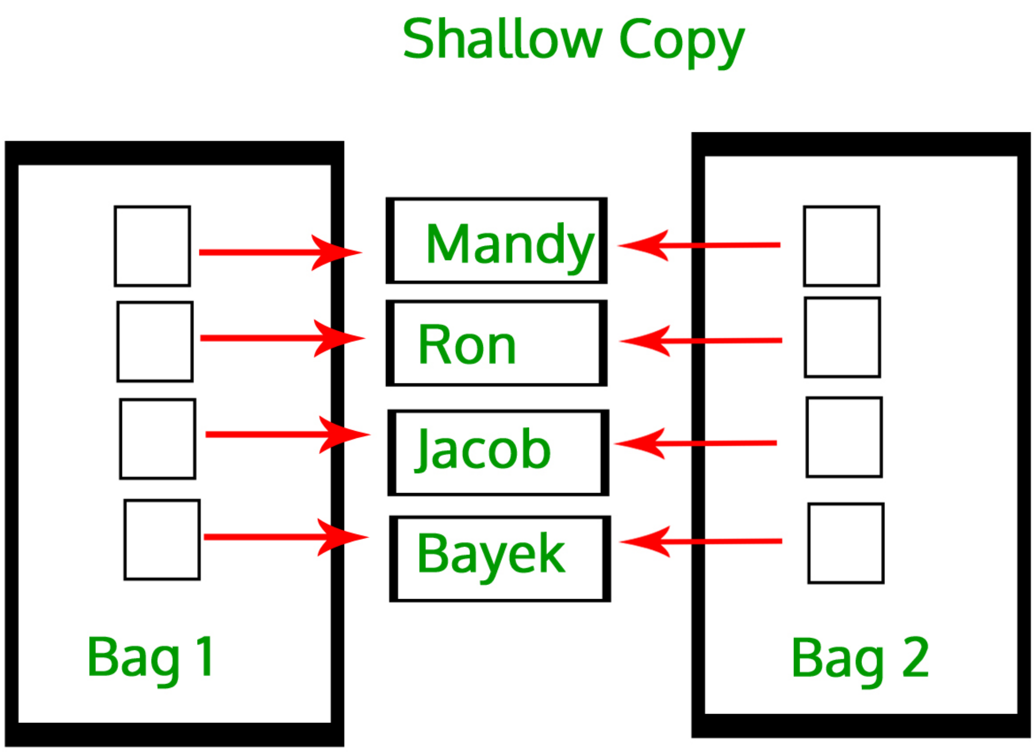
Lecture 17: Deep Copy

Shallow Copy & Deep Copy

Because 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 **dangling pointers**.

```
#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()
{
```

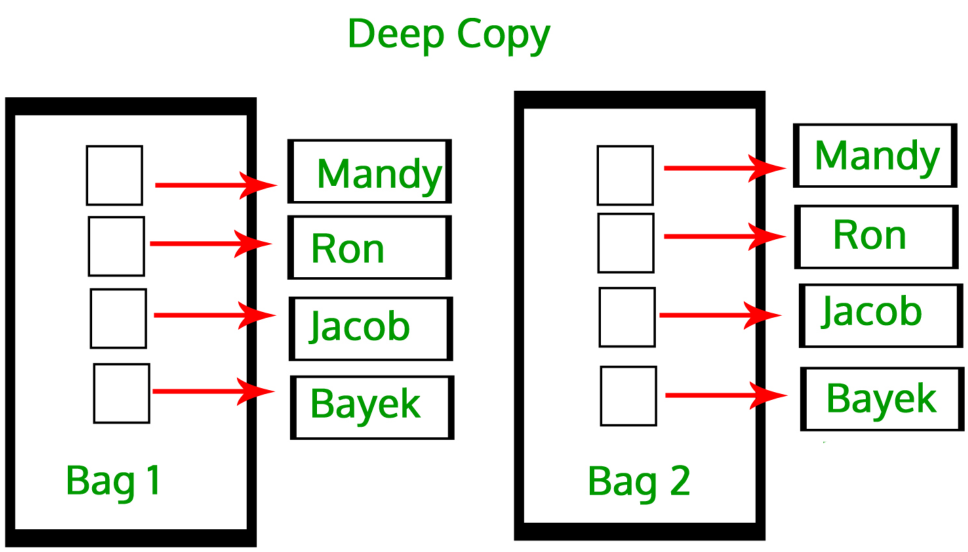
```
    Bag bag1;
    bag1.insert("VE280");
    Bag bag2 = bag1;
}
```

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 to destruct this 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.

> 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.

A reminder:

```
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
};

MyClass::MyClass(MyClass &that)
{
    if (this == &that){
        return;
    }
    else{
        destory(); // Destruct this
        // Do deep copy
    }
}

MyClass & MyClass::operator=(const MyClass &that)
{
    if (this == &that){
        return *this;
    }
    else{
        destory(); // Destruct this
        // Do deep copy
    }
}
```

These are 5 typical situations where resource management and ownership is critical. You should never leave them unsaid whenever dynamic allocation is involved. Traditionally **constructor/destructor/copy assignment operator** forms a rule of 3. Move semantics is a feature available after C++11, which is not in the scope of this course.

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;
```

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

- A destructor

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

- A copy constructor

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

- An assignment operator

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

Exercise

Recall binary tree and in-order traversal. We define that a good tree is a binary tree with ascending in-order traversal. How to deep copy a template good tree provided interface:

```
template <class T>
class GoodTree {
        T *op;
        GoodTree *left;
        GoodTree *right;
public:
    void removeAll();
    // EFFECTS: remove all things of "this"
    void insert(T *op);
    // REQUIRES: T type has a linear order "<"
    // EFFECTS: insert op into "this" with the correct location
    //                  Assume no duplicate op.
};
```

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

The sample answer is as follows.

```
template <class T>
void GoodTree<T>::copy_helper(const GoodTree<T> *t) {
    if (t == nullptr)
        return;
    T *tmp = new(t->op);
    insert(tmp);
    copy_helper(t->left);
    copy_helper(t->right);
}
```

```
template <class T>
void GoodTree<T>::copyAll(const GoodTree<T> &t) {
    removeAll();
    copy_helper(&t);
}
```

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.
`grow()` :

- 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];
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.

Learn more about amortized complexity in VE281/EECS281.