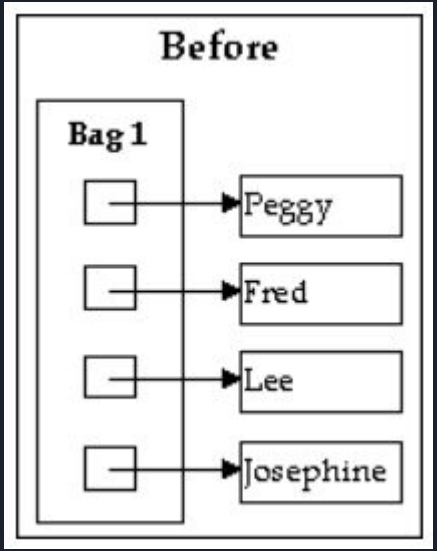
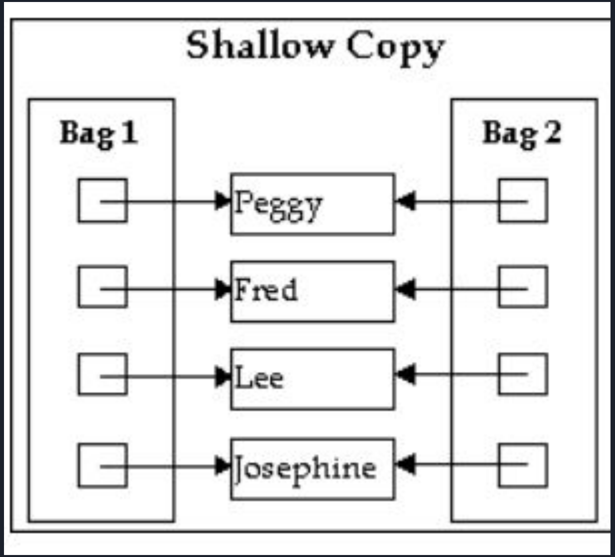
Exam 2 Review

Const

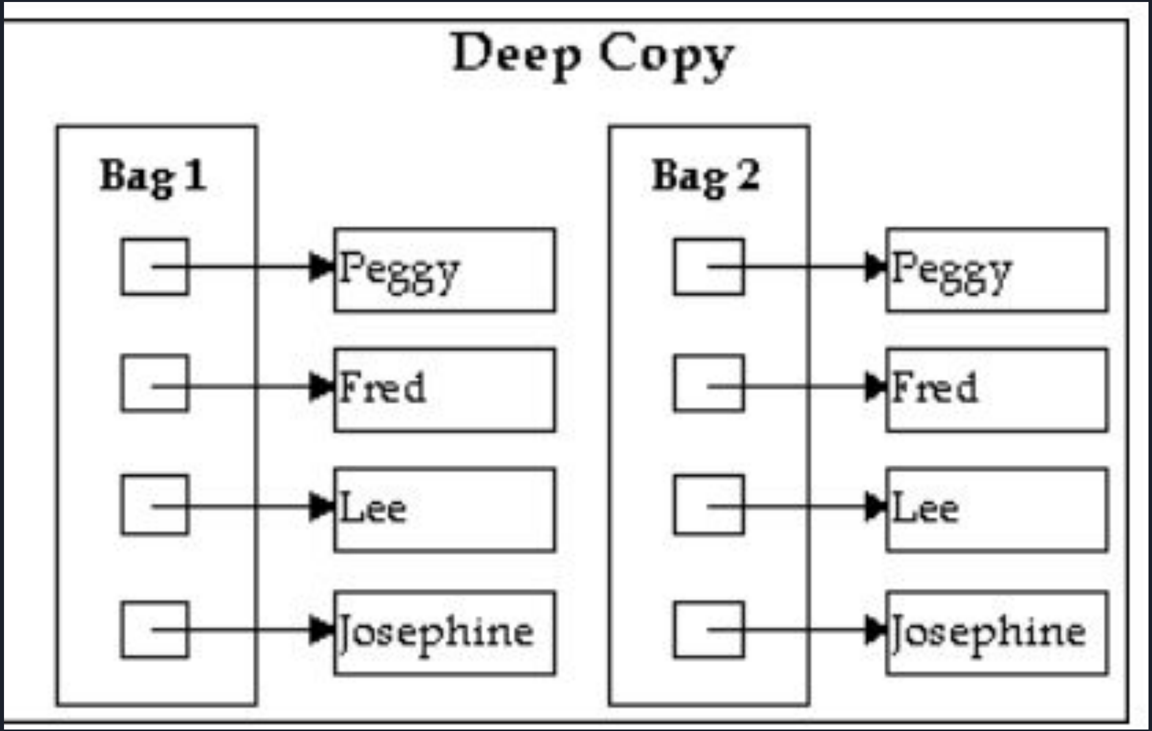
- Int* const pointer = &A → Pointer ITSELF is constant. The data it points to is NOT constant.
- Const int* const pointer = &A → Both pointer and data is constant
- Const int* pointer = &A → The pointer is NOT constant but the data IS.
- When a function is const, there are no changes to the object coming in.
- When a method is const there are no changes to *this* object.

Shallow Copy





Deep Copy


Deep Copy
Bag 1
Peggy
Fred
Lee
Josephine
Bag 2
Peggy
Fred
Lee
Josephine

Copy Constructor vs Assignment Operator

- Overall goal is same: copy one object into another
- **Copy constructor** can be called only ONCE, when an object is FIRST created
- **Copy assignment operator** may be called multiple times in object's lifetime
  - An object may have old data in it that needs to be cleared out before copying
  - With non-dynamic memory, this is a non-issue (just overwrite it)
  - With dynamic memory, you might need to **delete** previously allocated memory
- But aren't we duplicating a lot of code?...

What's an easy way to reuse

Put it in a function!

```
LineItem& LineItem::operator=(const LineItem& otherObject)
{
	name = otherObject.name;
	description = otherObject.description;
	quantity = otherObject.quantity;
	price = otherObject.price;
	return *this;
}
```

```
LineItem::LineItem(const LineItem &otherObject)
{
	name = otherObject.name;
	description = otherObject.description;
	quantity = otherObject.quantity;
	price = otherObject.price;
}
```

The Destructor

- A method which is called when an object is **destroyed**, either:
- When it **falls out of scope** (like a temporary variable in a function), or…
- When delete is called on a **pointer to an object**

- The purpose of a destructor is to clean up or "shut down" an object…
  - Delete any **dynamically allocated memory**
  - Notify another object/function that destruction has occurred
  - Print something out to the screen as a result of this object finishing its task

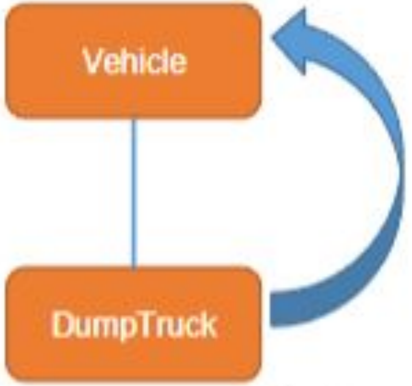
The destructor will ALWAYS have this format:

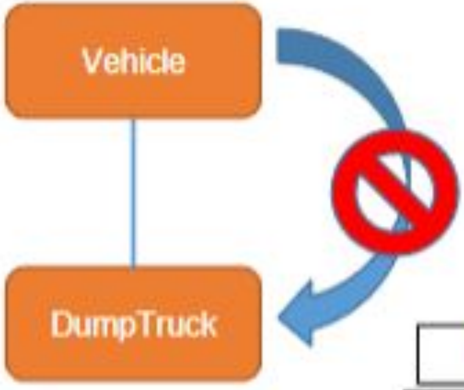
```
~ExampleClass(); // Prototype
ExampleClass::~ExampleClass() { } // Definition
```

The implicitly declared version of any destructor does… absolutely nothing.

Assumption is there is nothing to clean up (no **new** memory to **delete**)

Upcasting and Downcasting





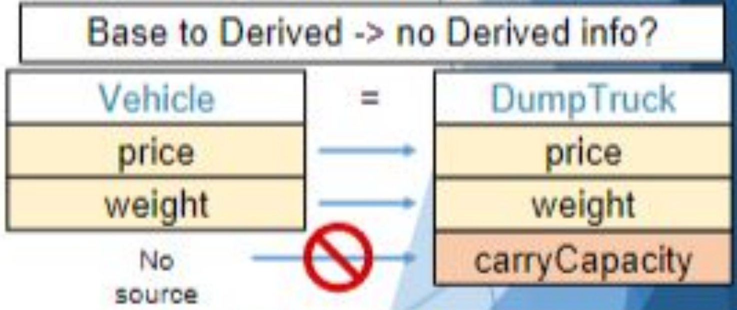
```
Vehicle vehicle;
DumpTruck dTruck;
```

```
vehicle = dTruck;
```

Upcasting works implicitly, because dTruck IS A Vehicle

```
dTruck = vehicle
```

Downcasting… doesn't work. Vehicle IS NOT A DumpTruck



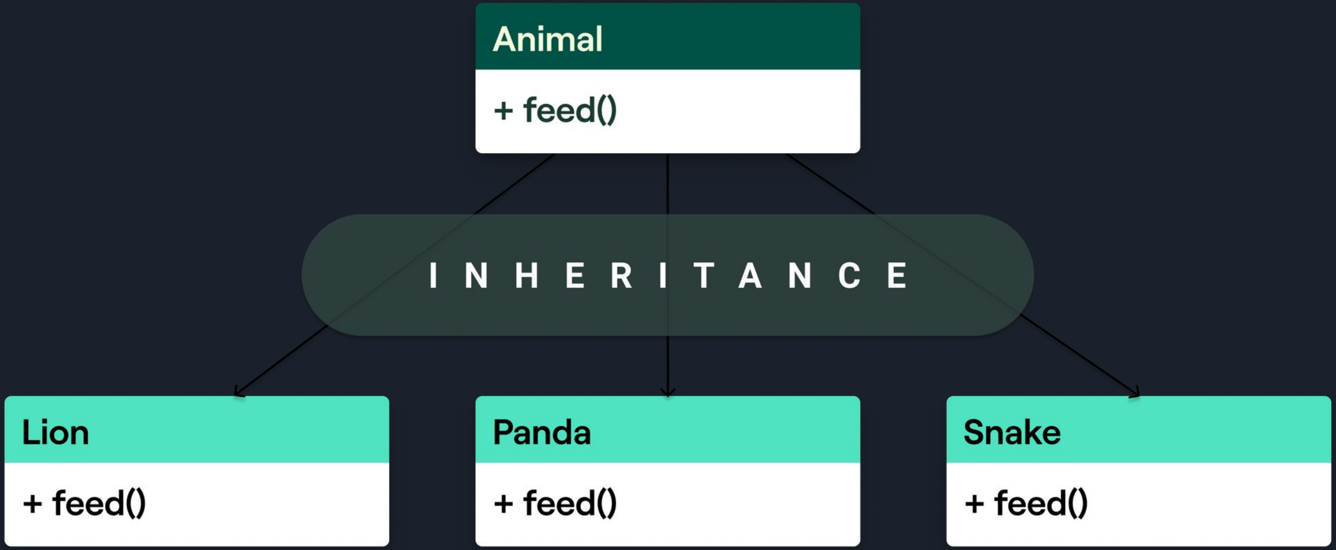
Base to Derived -> no Derived info?

Vehicle	=	DumpTruck
price	→	price
weight	→	weight
No source	🚫	carryCapacity

Where do we get a value for carryCapacity? Your program won't just make something up…

Abstract Base Classes

- This is when you want to make an *interface.*
- When you want a base class that is not directly used, but is derived from.



Can a destructor be left without a definition?