Exercise: Object Lifetime for Instances of Classes

Consider the following class definitions that Manjeet is writing:

```
class Cow {
 public:
    Cow() = delete;
    Cow(const Cow& other) = default;
    Cow(size_t maxFriends);

    void addFriend(std::string name);

 private:
    size_t numFriends_;
    size_t maxFriends_;
    Sheep** myFriends_;
};


class Sheep {
 public:
    Sheep() = default;
    Sheep(const Sheep& other) = default;
    Sheep(std::string name);
 private:
    std::string name_;
};
```

with the following definition for the Sheep's parameterized constructor:

```
Sheep::Sheep(string name)
    : name_{name}
{ }
```

How many constructors should Manjeet expect to write definitions for in `cow.cpp`?

Manjeet needs to write one constructor: the parameterized constructor.

How many total constructors will be available to users of Manjeet's `Cow` class?

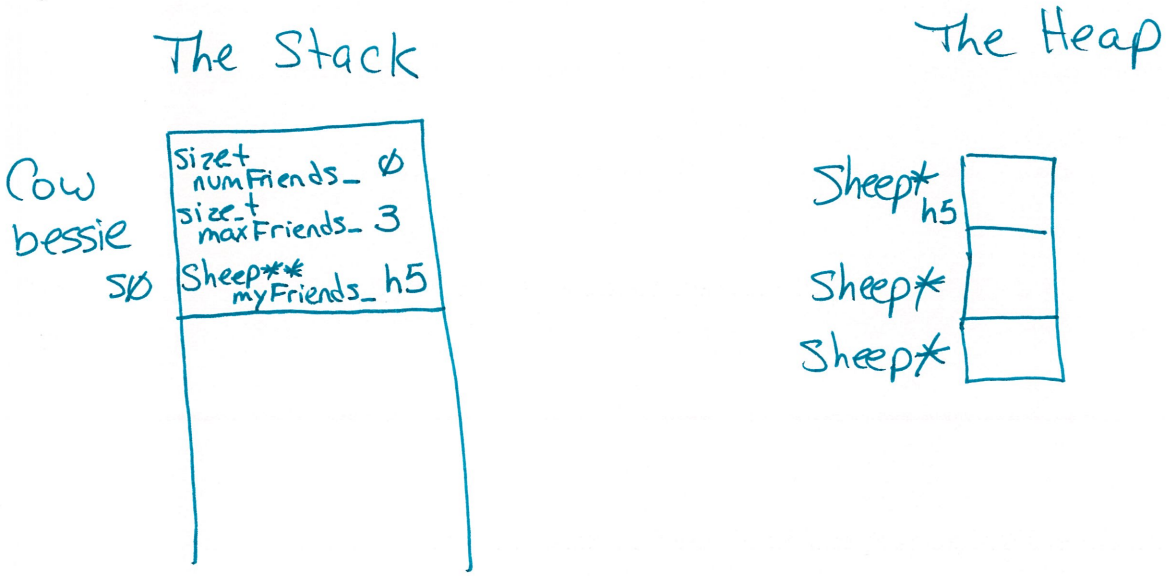
There will be two available `Cow` constructors: the one Manjeet writes and the synthesized copy constructor.

Manjeet's definition for the Cow class's parameterized constructor looks like this:

```
Cow::Cow(size_t maxFriends)
    : numFriends_{0},
      maxFriends_{maxFriends},
      myFriends_{new Sheep*[maxFriends]}
{ }
```

For the following code, draw what memory will look like *just before the closing curly brace of main.*

```
int main() {
    Cow bessie{3};
    return 0;
}
```



Implement the Cow class's addFriend(name) method, which should fill in an entry in the Cow's myFriends_ array with a Sheep named name – but only if the Cow does not already have maxFriends_ friends!

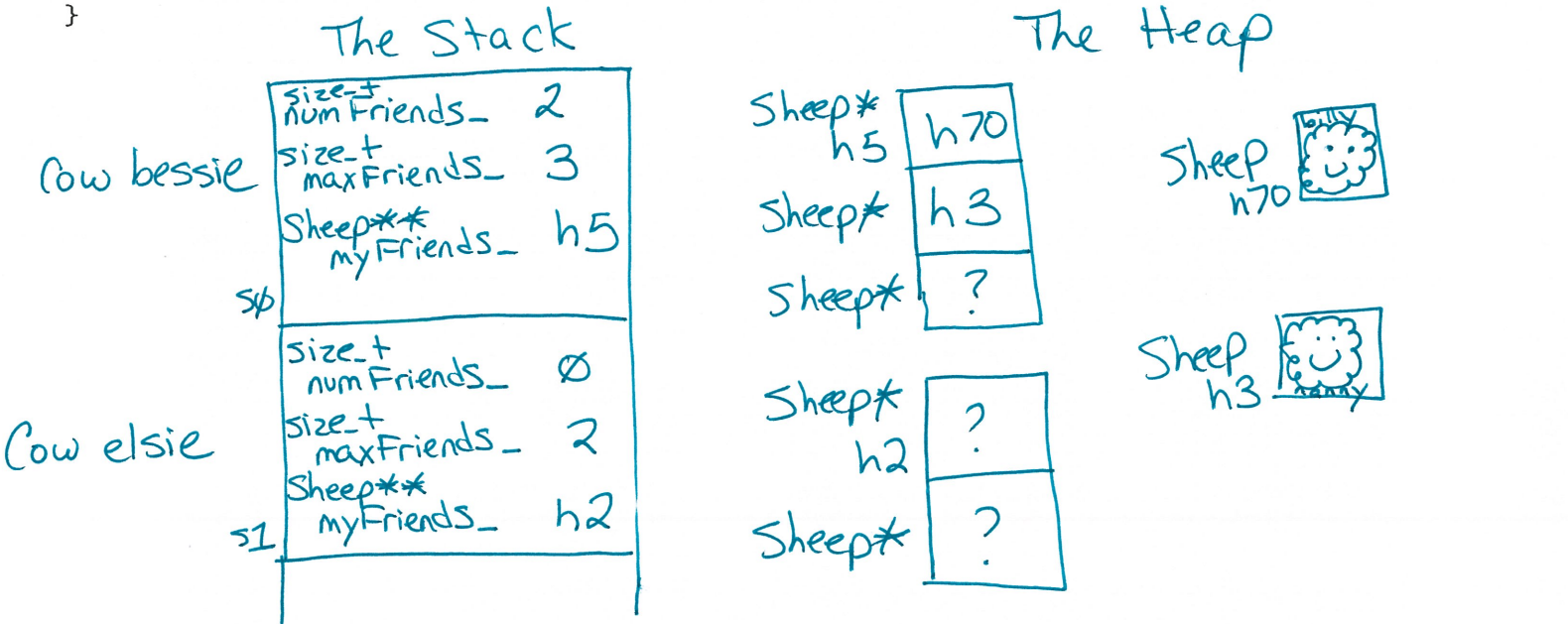
```
void Cow::addFriend(string name) {
    if (numFriends_ < maxFriends_) {
        myFriends_[numFriends_] = new Sheep{name};
        ++numFriends_;
    }
}
```

For the following code, draw what memory will look like *just before the closing curly brace of main.*

```
int main() {
    Cow bessie{3};
    bessie.addFriend("billy");
    bessie.addFriend("nanny");

    Cow elsie{2};

    return 0;
}
```



On your memory diagram, circle the memory that will be *leaked* when the `main` function returns.

The `Cow` class's *destructor* is where Manjeet should write the instructions that need to be run to clean up the memory that's being used by a `Cow`. That function is called `~Cow()`.

Help Manjeet write a destructor that will avoid the memory errors above.

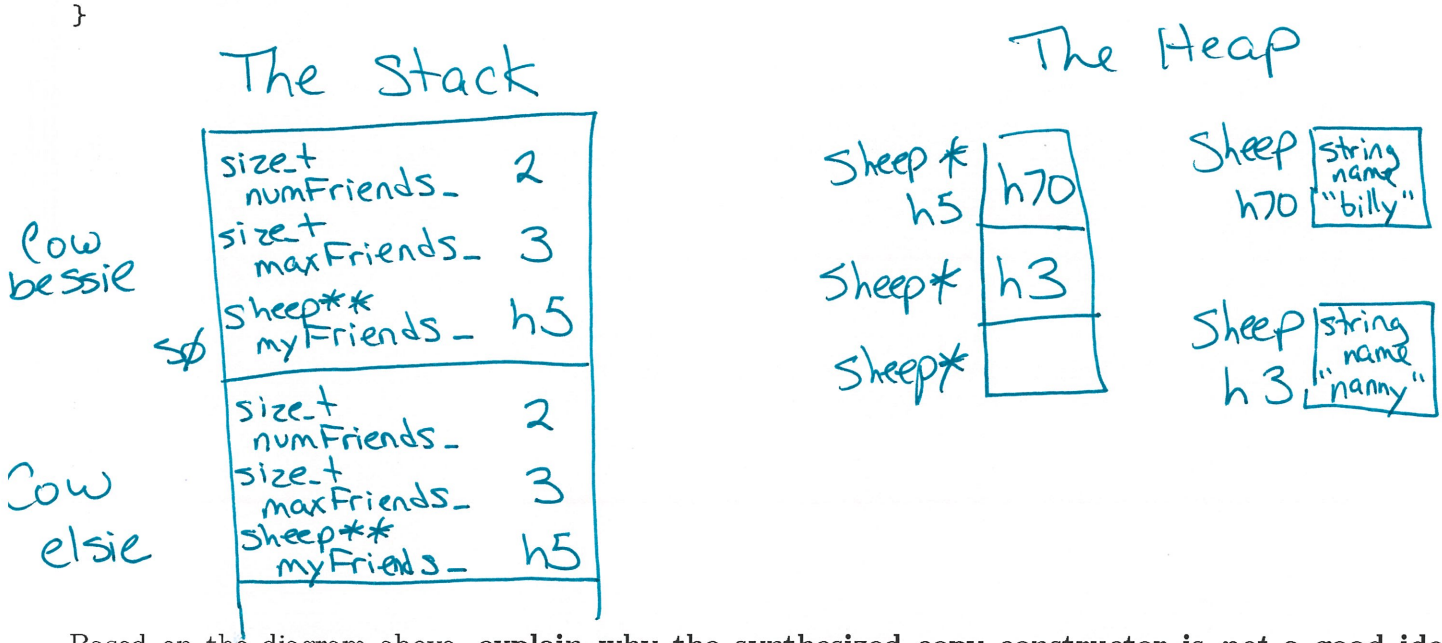
```
Cow::~Cow() {
    for (size_t i = 0; i < numFriends_; ++i) {
        delete myFriends_[i];
    }
    delete[] myFriends_;
}
```

The compiler's *synthesized copy constructor* will directly copy the values of each of the data members from one `Cow` to another.

Draw what memory will look like if Manjeet runs the following code:

```
int main() {
   Cow bessie{3};
   bessie.addFriend("billy");
   bessie.addFriend("nanny");

   Cow elsie{bessie};
}
```



Based on the diagram above, **explain why the synthesized copy constructor is *not* a good idea for Manjeet's `Cow` class.**

The synthesized copy constructor will copy the memory address in `bessie`'s `myFriends_` data member, which means that both `Cow`s will share the same array of `Sheep*`s. That means they'll both try to `delete` the same array in their destructors, which would result in a double delete.

Write a copy constructor for Manjeet.

```
Cow::Cow(const Cow& other)
    : numFriends_{other.numFriends_},
      maxFriends_{other.maxFriends_},
      myFriends_{new Sheep*[other.maxFriends_}
{
    for (size_t i = 0; i < numFriends_; ++i) {
        myFriends_[i] = new Sheep{other.myFriends_[i]};
    }
}
```

A class's *assignment operator* is the function that describes how an existing instance of a class should be assigned the value of a different instance of the class. It looks a lot like the copy constructor, but it plays a very different role – whereas the copy constructor tells how to *initialize* a new instance of a class, the assignment operator tells how to change an existing instance during *use*. Manjeet's assignment operator looks like this:

```
Cow& Cow::operator=(const Cow& other) {

    if (&other == this) {
        return;
    }

    for (size_t i=0; i < numFriends_; ++i) {
        delete myFriends_[i];
    }
    delete myFriends_;

    myFriends_ = new Sheep*[other.maxFriends_];

    for (size_t i=0; i < numFriends_; ++i) {
        myFriends[i] = new Sheep{other.myFriends_[i]};
    }

    myFriends_ = other.myFriends_; // Get rid of this line!
    numFriends_ = other.numFriends_;
    maxFriends_ = other.maxFriends_;

    return *this;
}
```

which Manjeet wants to use to write code like:

```
int main() {
    Cow bessie{3};
    Cow elsie{2};

    elsie = bessie;
}
```

... which should end with `elsie` looking identical to `bessie`. As it stands, though, the assignment operator will lead to some memory errors.

Describe the memory problem(s) that will occur. Drawing a memory diagram is not required here, but may be helpful.

As with the synthesized copy constructor, this will cause both `Cows` to share the same memory, which means they will both try to `delete` the same memory. Additionally, though, when `myFriends_` is set equal to `other.myFriends_`, we will lose the only way we had to keep track of `elsie`'s heap data – which means we've introduced a memory leak.

Correct the assignment operator above to fix the memory errors.

Exercise: Version Control

Ray and Yoshi are working together on a Clinic project for Cowtopia Inc. They've set up a repository (CowtopiaClinic) that they both have access to on GitHub to track their work. By default, GitHub repositories only have one file in them (README.md) when the repository is created.

Ray clones the repository and creates a new file in the called "ProjectTimeline.md" in Visual Studio Code:

```
1. Create timeline
```

Yoshi is excited to jump in and start too. Yoshi and Ray sit next to each other on their own laptops, and Yoshi runs the command `git clone https://github.com/ray-and-yoshi/CowtopiaClinic.git`

What file(s) will be in Yoshi's `CowtopiaClinic` directory?

README.md

For Yoshi to edit `ProjectTimeline.md`, what command(s) does Ray need to run? What command(s) does Yoshi need to run? Be sure to list the commands in the order that they need to be run, and to *justify* your answer.

Ray needs to ***add, commit,*** *and* ***push*** *the new file.*

Then, Yoshi needs to pull* the changes from GitHub.*

Yoshi and Ray successfully run the commands above, and now Yoshi is ready to edit `ProjectTimeline.md`. Yoshi, ever the optimist, changes the contents of `ProjectTimeline.md` to:

```
1. Create timeline (done)
```

and then saves the file.

What are the contents of `ProjectTimeline.md` on Ray's machine? On Yoshi's machine? On GitHub?

Ray's machine: `1. Create timeline`

Yoshi's machine: `1. Create timeline (done)`

GitHub: `1. Create timeline`

What command(s) do Ray and Yoshi need to run so that the changes will show up on GitHub? On Ray's machine?

Yoshi needs to ***add, commit,*** *and* ***push*** *the new file for the changes to show up on GitHub.*

For the changes to show up on Ray's machine, then Ray needs to pull* the changes from GitHub.*

Exercise: operator[]

We have seen in class that we can use the `[ ]` syntax with both statically-allocated and dynamically-allocated arrays. In both cases, the compiler treats the square brackets as for shorthand for pointer math and dereferencing.

As it turns out, we can define what that syntax should do for *any* class that we define. In code like:

```
MyVectorClass v{4};
std::cout << v[0] << std::endl;
```

the `v[0]` is translated by the compiler into a call to a member function with the name `operator[]`, which we might define like this:

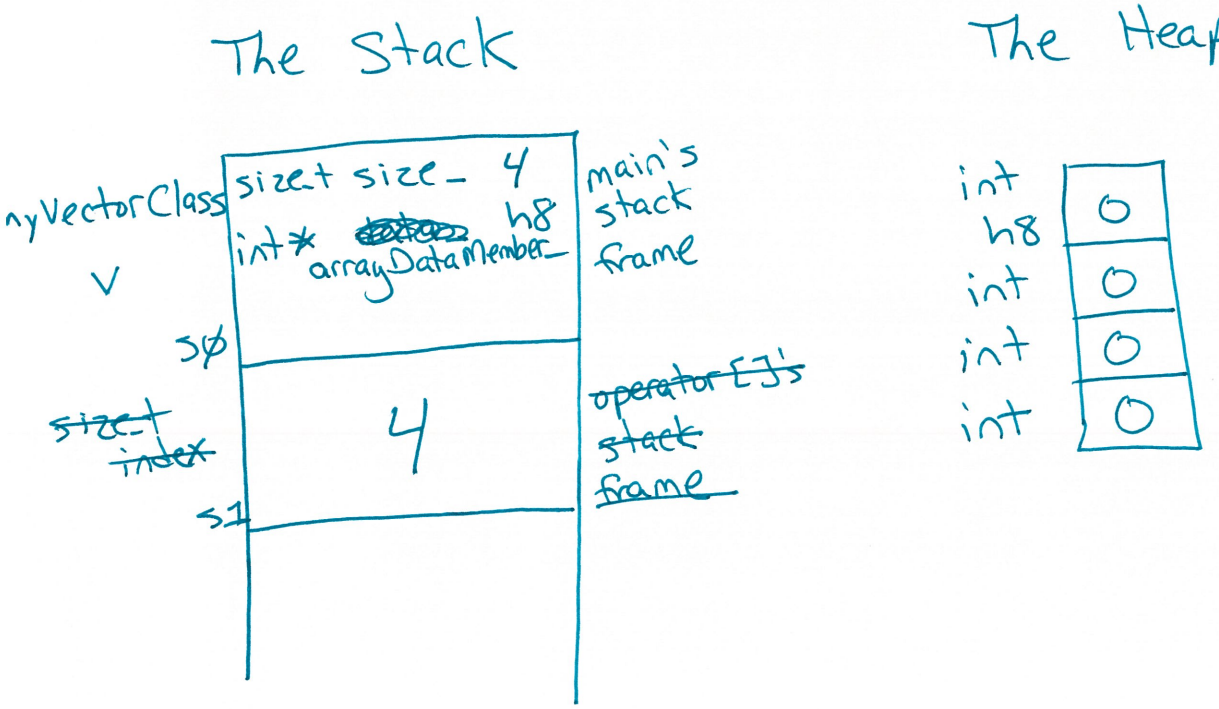
```
int& MyVectorClass::operator[](size_t index) {
    return *(arrayDataMember_ + index);
}
```

Assume that the *parameterized constructor* for the `MyVectorClass` class, `MyVectorClass(size_t size_)` creates a dynamically-allocated array the right size to hold `size_` integers and initializes all of its values to `0`.

What data members will a `MyVectorClass` object need to store? How much space will it take up on the stack?

A `MyVectorClass` object sill need to store two data members: the `size_` and a pointer to the array of `int`s on the heap.

Draw what the stack and heap look like when the code sample runs from above that creates a `MyVectorClass` object and then prints out its `0`th element.



Another thing we'd like to do with the `[ ]` syntax is to *change* a value in a `MyVectorClass` object.

Explain (using a memory model and prose) why this won't work with the current definition of `operator[]`:

```
MyVectorClass v;
v[1] = 70;
```

*This won't work because `operator=` will return a **copy** of the value in the array, not the spot in the array itself.*

We can fix the `operator[]` function definition to work in the case from (3) by using what we have seen in class about references.

Modify the function definition above to correct the behavior, and explain why your fix will get the desired behavior.

By changing the return type to a ***reference*** *to an int, we'll get another name for the element in* ***v*** *that we want to change. Then, when we assign to it, we'll change the data itself.*