Name: _____

Today's Date: _____

**Today's Goals**:

- Understand CS70 Exam Logistics
- Practice problems that are like CS70 exam questions
- Identify topics to review/practice before the exam.

**Today's Question(s)**

Which concept(s) are you most concerned about?

Which concept(s) have you mastered?

How will you prepare for the CS70 exam?

## Midterm Exam

- Take-home exam
  - Available this Friday, 5pm
  - Unlimited time to take it, but must finish it once you've started.
  - Due back 8am Tuesday morning
- Covers everything through today.
- One, double-sided 8.5" x 11" sheet of notes that **you hand-write yourself**.
- Asks you to demonstrate what you know *and* incorporate new ideas.

## Review Opportunities

- Friday's lab with Prof. Medero: Come to either or both!
- Review session with the grutors: Watch Piazza for announcement!

# Exercises: Practice Exam Problems

## Style, Elegance & Simplicity

You should be able to

- Discuss the value of good style, including
  - The impact (if any) of good style on program and programmer efficiency
  - The perils of "leaving style for later"
- Relate the following concepts to programming style
  - Elegance
  - Organization
  - Consistency
  - Idiom
  - Correctness
  - Extensibility
- Determine what aspects of a program require comments
- Place comments appropriately so that they are highly readable
- Devise appropriate variable names, based on context
- Convert code to use idiomatic looping constructs

# C++

### C++ Memory Model

You should be able to

- Contrast member initialization against assignment
- Express the memory layout of a program diagrammatically
- Describe and apply C++ scoping rules for local variables
- Determine when objects are allocated on the stack, and when on the heap
- Compare and contrast the heap and the stack
- Give a rationale for providing a stack as well as a heap
- Describe and apply **new** and **delete** for
  - Single objects
  - Arrays of objects
- Contrast and explain the rationale for both kinds of **new** / **delete**
- Describe and detect the following coding errors
  - Double deletion
  - Memory leaks
  - Pointer-to-object/pointer-to-array-of-object confusion
- Describe and use references
- Explain and contrast when it is appropriate to use each of the following techniques, and the lifetimes of the names and objects involved
  - Pass by value
  - Pass by constant reference
  - Pass by reference
- Apply and explain pointer arithmetic
- Use and explain primitive arrays

### Basic C++ Object Programming

You should be able to

- Explain and apply the technique used to disable copy constructors and/or assignment operators

### C++ Language Features

You should be able to

- Use the "Inside Out Rule" to determine the types of variables
- Use the C preprocessor to include source lines from other files
- Determine and describe when and where `const` should be used
- Resolve problems that may occur when `const` is used

## Program Development with Standard Unix Tools

### UNIX Tools

You should be able to

- Navigate using the UNIX command line
- Use `GitHub` for version control

### Compiling

You should be able to

- Enumerate and explain the stages of compilation
- Use GNU-style compiler tools (i.e., `clang++`, `g++`) to create
  - An executable program from a single source file
  - An object-code file from a single source file
  - An executable program from multiple-object code files
- Explain and create Makefiles that include
  - Necessary and sufficient description(s) of file dependencies
- Describe and apply the algorithm used by make to rebuild files based on dependencies

## 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`?**

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

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

```
Cow::Cow(size_t numSpots, 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) {









}
```

**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() {




}
```

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.**

**Write a copy constructor for Manjeet.**

```
Cow::Cow(const Cow& other)
    :


{









}
```

---

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;
    }









    myFriends_ = other.myFriends_;

    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.

**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?**

**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.**

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?**

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

## 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?**

**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;
```

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.**