Please watch or skim the videos listed on the Canvas assignment corresponding to this LGA. There is a lot more content than usual in this LGA, since it should be review material for many of you. Feel free to fast-forward though any bits you feel pretty confident about already.

## Individual self-test questions

**All students** should answer this set of questions. You will compare your answers with your group during the next lecture period. Note, we may from time to time include trick questions.

- Roughly how much memory is required to store an object?

- How would you allocate memory for an object of type `foo` on the stack?

- How would you allocate memory for an object of type `foo` in the free store?

- Last two questions again, assuming you need to pass in an integer argument (0, perhaps).

- How would you allocate memory for an array of `foo` of size 5 on the stack? On the free store?

- Write down function signatures for an overload of the multiplication operator acting on two objects of type `foo` and returning a `foo`, in two ways:

- Write down the function signature for the copy constructor for class `foo`:

- Write down the function signature for the assignment operator overload (as part of the "big 3") for class `foo`:

- The last question again, but assuming `foo` is a template class taking a single type parameter:

- What are two examples of when the copy constructor is called *implicitly*?

- What are two ways to invoke the copy constructor *explicitly* for class `foo`?

- What is the difference between copying and assigning an object?

- What is the difference between a shallow copy and a deep copy?

## Group questions

Distribute the following questions across the members of your group. You will share your solutions (and most importantly the *method* of your solutions) during the next lecture period. Divide up the questions so that each question has at least three solutions from different group members.

1. (Intermediate) What does the following code print to the standard output?

```
class point {
public:
    int x, y;
    point() { x = 0; y = 0; }
    point(int d1, int d2) { x = d1; y = d2; }
};

point operator+(point t, point u) { return point(t.x + u.x, t.y + u.y); }
ostream& operator<<(ostream& out, point p) {
    out << "(" << p.x << ", " << p.y << ")";
    return out;
}

template <typename T>
void print_sum(T* arr, unsigned int len) {
    if (len == 0) return;
    T acc = arr[0];
    for (unsigned int i = 1; i < len; i++) {
        acc = acc + arr[i];
    }
    cout << acc << endl;
}

int main() {
    int iarr[4] = {1, 2, 3, 4};
    print_sum(iarr, 4);

    string sarr[3] = {"blue", "red", "yellow"};
    print_sum(sarr, 3);

    point parr[2];
    parr[0].x = 10;
    parr[0].y = 20;
    parr[1].x = -5;
    parr[1].y = 17;
    print_sum(parr, 2);

    return 0;
}
```

2. (Intermediate) Below is `fraction.h` which has the declaration of the `fraction` class. The `fraction` class stores a numerator and denominator in dynamically allocated integers.

   Fraction.h:

```
class fraction {
private:
    int* num = nullptr;
    int* den = nullptr;
public:
    // constructor
    fraction(int n, int d = 1) {
        num = new int(n);
        den = new int(d);
    }

    // big 3
    fraction(const fraction& orig);
    fraction& operator=(const fraction& orig);
    ~fraction();

    // arithmetic operations
    fraction operator+(const fraction& rhs);
    friend fraction operator*(const fraction& lhs, const fraction& rhs);

    // printing
    friend ostream& operator<<(ostream& out, const fraction& f);
};
```

   Complete the fraction class by implementing the "big 3" methods, arithmetic operator overloads (just addition and multiplication, no reduction required), and `ostream` output operator. (Note we have deliberately set this up so you must implement one arithmetic operation as a method of `fraction`, and the other as a free function.) Divide the work up among your group members.

   Below is a some test code for your fraction class:

```
int main() {
    fraction a(5, 8);
    fraction b(4, 9);
    fraction c = a;
    cout << (a * b) << endl; // prints 20/72
    cout << (a + b) << endl; // prints 77/72
    cout << (b * c) << endl; // prints 20/72
    a = c = b;
    cout << (a * b) << endl; // prints 16/81
    return 0;
}
```

3. (Basic) Implement the template function `print_five()`, which takes in a single value and prints the value 5 times (using `cout`). Write and test code that uses the template successfully with at least 3 different types.

4. (Basic) Write code to demonstrate what happens when you use the above template with an argument for which there is no defined print behavior. (Write some code and feed it to the compiler to see the error message.)

5. (Intermediate) Write code for a template class that can *contain* exactly 5 items of any type, which will be supplied via the constructor. You can store the values internally however you wish, but you must include a getter method that will return a value when supplied the integer index of the value.

   E.g., suppose your class is named `holds5`. Then code like the following should work:

```
void holds5_test()
{
    holds5<string> holder("zero", "one", "two", "apple", "four");
    holder.get_item(3);    // returns "apple"
}
```