Assignment 1
Feedback
In Assignment 1, the Patient class overloads the `==` operator and the `>` operator (see next slide).

This signifies that it is the responsibility of the Patient class to:

- Determine when two Patient objects are equal, and
- Establish when Patient object A > Patient object B

It is not the responsibility of the List class:

- This is to say that List class should not refer to any of the Patient class data members (even by calling its getter).
- In other words, we should not use/see `getCareCard()` in the code of List.cpp.
Since we were allowed to add more methods to the Patient class, I also overloaded the `<` operator.
Patient Class - Constructors

We were asked to add more parameterized constructors.
So, I added a parameterized constructor with 5 parameters, one for each of the Patient class’ data members:

```
// Default Constructor
// Description: Create a patient with a care card number of "0000000000".
// Postcondition: All data members set to "To be entered",
// except the care card number which is set to "0000000000".
Patient::Patient() : name("To be entered"), address("To be entered"),
    phone("To be entered"), email("To be entered") {
    setCareCard("0000000000");
    std::cout << "Patient::default constructor" << endl;  // For testing purposes ... and curiosity
}

// Parameterized Constructor
// Description: Create a patient with the given care card number.
// Postcondition: If aCareCard does not have SIZE_OF_CARECARD digits (10),
// then care card is set to "0000000000".
// All other data members set to "To be entered".
Patient::Patient(string aCareCard) : name("To be entered"), address("To be entered"),
    phone("To be entered"), email("To be entered") {
    setCareCard(aCareCard);
}

// Description: Create a patient with the given name, address, phone number, email and care card number.
// Postcondition: If aCareCard does not have 10 digits, then care card is set to "0000000000".
Patient::Patient(string aName, string anAddress, string aPhone, string anEmail, string aCareCard) :
    name(aName), address(anAddress), phone(aPhone), email(anEmail) {
    setCareCard(aCareCard);
}
```
Patient Class – `setCareCard(...)`

... and created a `private` method called `setCareCard(...)` (its name sounds like a setter, but it isn’t as it is not a public method)

This private method ensures that the care card used to construct a Patient object (passed as a parameter to the parameterized constructor) has 10 characters and that each of these characters is a digit:

```cpp
// Description: Sets the patient's email - Private method
void Patient::setCareCard(const string aCareCard) {
    bool badCareCard = false;
    
    careCard = "0000000000";
    if (aCareCard.length() == SIZE_OF_CARECARD )
        for (int digit = 0; !badCareCard && digit < SIZE_OF_CARECARD; digit++)
            if ( !isdigit(aCareCard[digit]) ) badCareCard = true;

    if ( !badCareCard ) careCard = aCareCard;
    return;
}
```
Patient Class – Class Invariant

• Why are we validating the care card number this way?
• Because the Patient class has the following class invariant:

/*
 * Patient.cpp
 */

* Class Description: Models a walk-in clinic patient.
* Class Invariant: Each patient has a unique care card number.
  This care card number must have 10 digits.
  This care card number cannot be modified.

• And we want to make sure that each instantiated Patient object satisfies its class invariant by having a care card number if 10 digits
Patient Class – Validate CareCard

- However, the most efficiency way to ensure that every instantiated Patient object satisfies its class invariant by having a care card number if 10 digits is by ensuring that the user (receptionist) enters a valid care card number
  - And this would be done as part of the walkIn.cpp program (code) – see last slide
List Class

- In order to satisfy Requirement 7.
  - “The print method of the List class must have a time efficiency of $O(n)$ where $n$ is the number of elements in the List.”

and its class Invariant

- “Each element is unique (no duplicates).”

`insert(...) must behave as follows:`

- Once it has ascertained that a Patient object has not already been inserted, `insert(...) must either`
  - insert each element in descending sort order OR
  - sort all Patient objects once it has inserted a Patient object in the most efficient fashion -> at `elementCount` (i.e., at the end of the section of the array already containing Patient objects)
List Class

- **insert(...)**
  - May have to shift Patient objects in the array in order to make space for the new Patient object to be inserted

- **remove(...)**
  - May have to shift Patient objects in the array in order to overwrite the Patient object to be removed
List Class – Responsibility Issue!

Why? Because, it is not the List ADT class’ responsibility to know what makes two Patient objects equal. That is the responsibility of the Patient class.

Solution: So, we must ask the Patient objects if they are equal and we do this as follows (by using the overloaded == operator of the Patient class):

```cpp
if (target == element[i])
```

Should not be used!
However, this new statement

```cpp
if (target == element[i])
```

does not compile!!! 😞

Indeed, we get this error message:

```
List.cpp: In member function ‘Patient* List::search(const Patient&)’:
List.cpp:117:36: error: passing ‘const Patient’ as ‘this’ argument of ‘bool Patient::operator==(const Patient&)’ discards qualifiers [-fpermissive]
   if ( target == elements[index] ) {
              ^
make: *** [List.o] Error 1
```
List Class – Responsibility Issue!

**Why** does this statement not compile?

\[
\text{if (target == element[i])}
\]

**Answer:** When the code is compiled, the right hand side operand `element[i]` above is matched to the parameter `rhs` (*right hand side*) of the overloaded `==` operator:

```cpp
// Overloaded Operators
// Description: Comparison operator. Compares "this" Patient object with "rhs" Patient object.
// Returns true if both Patient objects have the same care card number.
bool operator == (const Patient & rhs);
```

And this parameter has the type `const Patient&`. In other words, it is a reference to a `Patient` object that cannot be modified (i.e., `const`). The left hand side operand `target`, on the other hand, is of type `Patient` object. How do we know this? Well, let’s have a look at the implementation of this overloaded operator on the next slide.
As we can see, the left hand side operand to `==` is composed of `this`, which is a pointer to a Patient object. As we talked about in class, a pointer is not the same thing as a reference.

So, how do we fix this and make the code compile? See next slide!
List Class - Solution

We fix the situation and make the code compile by switching the operands so that \texttt{target}, i.e., the \texttt{const Patient\&} is on the right hand side of the operator:

```cpp
// Description: Search for target element.
// Returns a pointer to the element if found, otherwise, returns NULL.
Patient* List::search(const Patient& target) {  // Linear or binary search
    Patient* result = NULL;
    bool found = false;

    for (int index = 0; index < elementCount && !found; index++) {
        if (elements[index] == target) {
            result = &elements[index];
            found = true;
        }
    }
    return result;
} // end of search
```
List Class – `printList(...)`

- Also, it is not the List ADT class’ responsibility to print the data members of Patient object.
- Printing itself (i.e., its data members) is the responsibility of a Patient object:

```cpp
// Description: Prints all elements stored in List by descending order of care card numbers.
void List::printList() {
    for (int index = 0; index < elementCount; index++) {
        cout << elements[index] << endl;
    }
    return;
}
```

- As you can see in the above code, the Patient class’ friend overloaded operator `<<` is used to print Patient object within the List class’ `printList()` method.
  - In a sense, you can imagine the List saying to each of the Patient objects: print yourself (because you know how).
List Class – return

On the Good Programming Style (GPS) web page of our course web site, it says: “In general, a function/method must only have one (unique) return statement…”

If we compare the above two implementations of the search method, we observe that the one on the left has 2 return statements and the one on the right, only one, located at the end of the method. This is to say that return is not used to exit the loop (as in the first implementation). Instead, a flag found is used to exit the loop. Also, notice the compound condition of the for loop.
walkIn.cpp

- Responsible for user interaction
- Possible Algorithm:
  
  At the start of the main() function
  
  - instantiate an array of Patient objects
    - This calls the Patient class default constructor for each cell of the array
  
  - when the receptionist selects "create a patient"
    - walkIn.cpp asks the receptionist for name, address, phone, email, care card #, etc...
    - then walkIn.cpp creates a Patient object with all this info (and this is why we are asked to create more constructors – one of them can be a 5-parameter constructor)
    - then walkIn.cpp calls the List’s insert method with this Patient object
    - as opposed to the algorithm on the next slice …

This is the object oriented way of creating a Patient
walkIn.cpp – Possible Algorithm

- when the receptionist selects "create a patient"
  - walkIn.cpp asks the receptionist for a care card #
  - then walkIn.cpp creates a Patient object with this care card # using the already defined parameterized constructor
  - then walkIn.cpp calls the List’s insert method with this Patient object
  - then walkIn.cpp asks the receptionist for the patient name and use the setName() setter to modify this Patient object's name
  - walkIn.cpp asks the receptionist for the patient address and use the setAddress() setter to modify this Patient object's address
  - walkIn.cpp asks the receptionist for the patient phone and use the setPhone() setter to modify this Patient object's phone
  - walkIn.cpp asks the receptionist for the patient email and use the setEmail() setter to modify this Patient object's email

This is ***not*** the object oriented way of creating a Patient
walkIn.cpp

- Suggestion: as it reads the user input (such as the care card number), it can validate this input
  - This way, it can ask the user (receptionist) to re-enter an invalid care card number
  - This is much more efficient than having the Patient class constructor setting it to 0000000000 and having the user (receptionist) modifying it later on