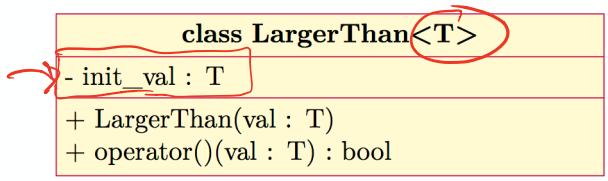
10. A function object is a construct that allows an object to be invoked with similar syntax to that of an ordinary function call; it provides us with the facility for function-like interaction with objects that are able to carry data as needed to perform some task. In this question, you are tasked to:

    (a) Write the declaration for the class template named `LargerThan` that has a single type parameter `T`. The structure of this user-defined type is detailed in the following UML diagram:



    i. (6 points) Write the class definition for the `LargerThan` class to the specifications in the UML diagram above:

```
template <typename T>
class LargerThan {
Public:
   LargerThan(T);
   bool operator()(T);

Private:
   T init_val;

}
```

ii. (6 points) Write the definition for the overloaded function call operator (`operator()`). This overloaded function will be passed a single argument to initialize the parameter; the function body of the overloaded function call operator will compare the actual argument passed to that parameter against `init_val`. If the value of the parameter is greater than `init_val`, return `true`; else return `false`.

```
template <typename T>
bool LargeThan<T>::operator()(T val)
{
    return val > init_val;
}
```

(b) (3 points) Illustrate how you would instantiate this class template with an integer template argument to create an object of type `LargerThan<int>` named `lt`, with an integer value of `94` passed to the parameterized constructor.

```
LargerThan<int> lt(94);
```

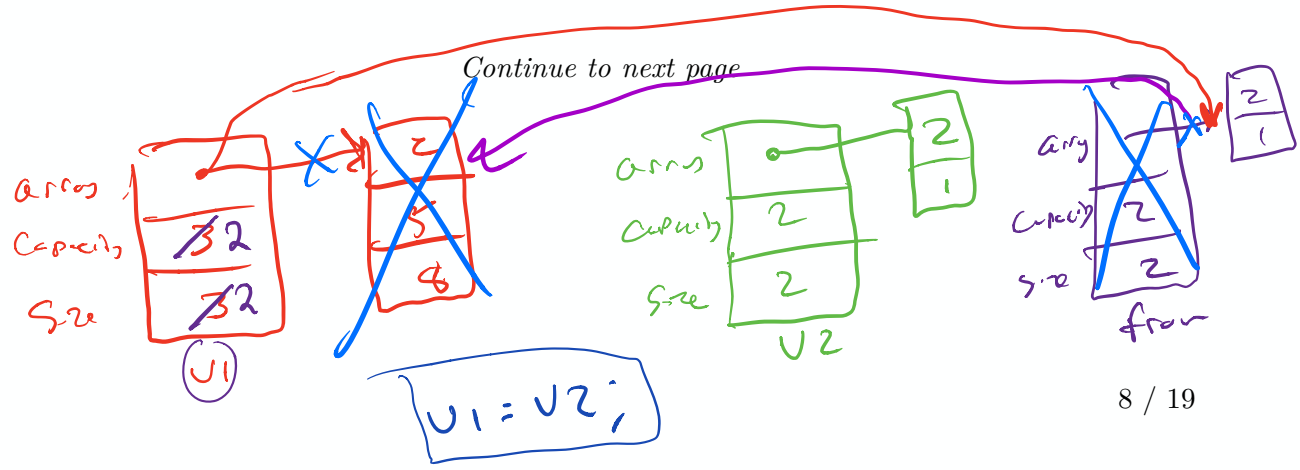
(c) (3 points) After initializing the object `lt`, how would you invoke the `operator()` to evaluate whether a passed integer value, say `23`, is larger than the value stored in `lt` s `init_val`?

```
lt(23);
```

Begin question 15...

```
Vec v0 = v2;
```

Continue to next page



15. One way to implement the copy assignment operator for a class is to use something called the copy and swap idiom. Using this idiom requires: a working copy-constructor, a working destructor, and a swap function that swaps out the old data for the new. The definitions needed to answer the following questions are provided below, including the copy assignment operator that uses the copy-and- swap idiom:

```
DynamicIntArray::DynamicIntArray(DynamicIntArray const& source) :
    max_size{source.capacity()},
    sz{source.size()},
    array{new int[source.capacity()]}
{
    for (decltype(source.size()) i = 0; i < source.size(); ++i) {
        array[i] = source.at(i);
    }
}

DynamicIntArray& DynamicIntArray::operator=(DynamicIntArray const&
    source)
{
    if (this != &source) {
        this->swap(*this, source);
    }
    return *this;
}

void DynamicIntArray::swap(DynamicIntArray &to, DynamicIntArray from) {
    to.max_size = from.capacity();
    to.sz = from.size();
    auto temp = to.array;
    to.array = from.array;
    from.array = temp;
}

DynamicIntArray::~DynamicIntArray()
{
    delete [] array;
}
```

*this → v1
source → v2
v1 = v2;
DynamicIntArray(source);
to another name for v2 / *this

(a) When the copy assignment operator is called by a `DynamicIntArray` object with a `DynamicIntArray` object as its argument, we first perform a self-assignment test.

i. (2 points) Why is this check needed in the copy assignment operator definition but not in the copy constructor?

(b) If it is not a self-assignment, we call `DynamicIntArray::swap`.

i. (2 points) What is the type of each parameter of `DynamicIntArray::swap`?

ii. (2 points) How are the parameters in DynamicIntArray::swap initialized? Specify whether each parameter is a reference to a DynamicIntArray or if it is a copy to a DynamicIntArray.

iii. (2 points) Is the copy constructor or the copy assignment operator used to initialize the parameter `from` with its argument? Recall that the semantics of argument passing are identical to the semantics of initialization.

(c) Inside the function body of `DynamicIntArray::swap`, `to.array` is swapped with `from.array`. At this point, we've successfully assigned the contents of `from` into `to`.

i. (5 points) Why does this procedure `DynamicIntArray::swap` result in a deep copy and not a shallow copy?

ii. (5 points) Why doesn't `DynamicIntArray::swap` result in a memory leak?