Data Structures and Objects
CSIS 3700

Lab 7 — Dynamic Memory

Goal

Create two classes, each using dynamic memory for storage.

Motivation

There are many times where static storage — memory set up by the compiler before the program runs — is not appropriate or possible to use.

For example, in Java, all non-basic data types must use dynamic storage. Another example is when you know the size an array needs to be when you are about to use the array, but you do not know the size when writing the program.

In these situations, dynamic memory provides a solution.

How It Works

Suppose our program needs to store a list of data in an array. The number of elements is an input to the program, and could be anywhere from 10 elements to 1000000 elements.

You would not want to allocate an array of 1000000 elements, since that would likely be wasteful. You also would not want to allocate a smaller array, since it is possible that more space would be needed.

The solution is to use dynamic memory. Instead of an array, declare a pointer to your data. For example, instead of

```
DataType          // the type of data you are storing
  data[1000000];
```

you would have

```
DataType
  *data;
```

Then, once you know the number of elements you will need, you can allocate space for the array:

```
data = new DataType[nElements];
```

where `nElements` is the variable containing the number of elements to be stored.

Once allocated, the pointer can be used just like an array, using [ ] to specify a position within the array.

When you are done with the space, you *must* give it back; C(++) does not have a garbage collector, so you must explicitly deallocate the space. This is done by the statement

```
delete[] data;
```

▹*Caveats*

- Do not use a pointer like an array before creating the space. This will crash your program.
- Once allocated, you must always keep a pointer pointing to the start of the space. Failure to do so will create a *memory leak*, which is a region of memory that cannot be deallocated and cannot be used.
- Always deallocate via `delete[]` when done. Failure to do so is another form of memory leak.
- Array bounds are not checked, just like they are not checked for normal arrays.

Example 1 — A Sampler Class

Suppose you had a collection of items and you wanted to randomly select one, then randomly select another, and another, and so on. An element can only be selected once. How can this be done efficiently?

One solution is to assign a boolean flag to each item, initialized to false. When an item is selected, change the flag to true. To select, randomly pick a position and repeat until the selected flag is false. This works, but is not terribly efficient.

The following algorithm presents a much more efficient approach:

Algorithm 1 Sampling without replacement

Preconditions *elements* is a list of integers; $n > 0$

Postconditions Random element from list is removed and returned; n is decremented

```
1: procedure SampleNoReplacement($elements, n$)
2:     $i \leftarrow$ random integer with $0 \le i < n$          ▹ Select random position in the list

3:     $e \leftarrow elements[i]$                                ▹ Remember the selected item
4:     $n \leftarrow n - 1$                                      ▹ Decrement $n$
5:     $elements[i] \leftarrow elements[n]$                      ▹ Move last item into selected position

6:     return $e$
7: end procedure
```

The algorithm selects one item at "random," then takes the last item in the list and moves it into the position vacated by the selected item. This keeps the remaining items in a (smaller) contiguous list. This is very efficient; each selection takes $O(1)$ time.

Create a `Sampler` class with the following methods:

- `Sampler(uint32_t n)`
  Creates an array of `n` integers. Set `array[i]=i` for all slots. Store `n` in a class variable.
- `~Sampler()`
  Deletes the array.
- `uint32_t getSample()`
  Returns one element from the array using the algorithm given above. If `n` is 0, throw an `underflow_error` exception.

Test it out; create a `Sampler` object with 20 elements. Then use a loop to get 20 samples. Verify that each number 0 – 19 appears once as a sample.

Example 2 — Disjoint-Set Structures

Disjoint sets are a very cool structure, and extremely easy to implement. A disjoint set only supports two operations: a *union* operation that joins two disjoint sets into one, and *find* which picks one element from a disjoint set. As long as a particular set does not change, the *find* operation always returns the same element. The *find* operation also gives the same answer for any element in the same set.

The algorithms for union and find are given below.

Algorithm 2 Disjoint set union

Preconditions *elements* and *rank* are a disjoint set with elements a and b

Postconditions The disjoint sets containing a and b have been combined into one set

```
procedure DisjointSetUnion($elements$, $rank$, $a$, $b$)
    $a \leftarrow$ DisjointSetFind($a$)                        ▷ Get representatives for $a$ and $b$
    $b \leftarrow$ DisjointSetFind($b$)

    if $a \neq b$ then                                         ▷ Only union if $a$ and $b$ are in different sets
        if $rank[a] < rank[b]$ then                            ▷ Set with lower rank merged into set with larger rank
            $elements[a] \leftarrow b$
        else
            if $rank[a] = rank[b]$ then                        ▷ In case of tie, increment one set's rank
                $rank[a] \leftarrow rank[a] + 1$
            end if
            $elements[b] \leftarrow a$
        end if
    end if
end procedure
```

Algorithm 3 Disjoint set find

Preconditions *elements* and *rank* are a disjoint set with element a

Postconditions Returns the representative for the set containing a

```
procedure DisjointSetFind($elements$, $rank$, $a$)
    if $elements[a] \neq a$ then                               ▷ Connect $a$ directly to top of intree
        $elements[a] \leftarrow$ DisjointSetFind($elements[a]$)
    end if
    return $elements[a]$                                       ▷ Return top of intree
end procedure
```

Make a `DisjointSet` class. It should have pointers to integers for both arrays. It should have the following methods:

- `DisjointSet(n)`
  Create the `elements` and `rank` arrays, each with `n` elements. Initialize `elements[i]=i` and `rank[i]=0` for all slots.
- `~DisjointSet()`
  Deletes the arrays.
- `find(int a)`
  Performs Algorithm 3 to find the top of the intree containing a.
- `join(int a,int b)`
  Performs Algorithm 2 to join a's and b's sets together.

Once you've created your class, compile it along with the driver `main.cpp` in the repository and run the resulting program. The output should look similar to the sample output file that is in the repository.

What to turn in

Please turn in your source code as a single compressed file.