Name: ____

Today's Date: ____

Today's Goals

- Explain the *purpose* of templates
- Convert a non-templated class/function to be templated.

Today's Question(s)

If you find yourself writing the same several lines of code over and over, it's probably a good idea to encapsulate that code in a ____

Lingering Questions

Repeated lines of code

```
int x = 3;
int y = 4;
int z = (x + y) / (x - y) // important calculation

// ... later ...
int a = 2;
int b = -5;
int c = (a + b) / (a - b) // important calculation

// ... later ...
int q = 200;
int r = 1000;
int s = (q + r) / (q - r) // important calculation

// ... and so forth and so on ...
```

How can we make *this* better?

```
void printValueOfInt(int x) {
    cout << "The value is " << x << endl;
}
void printValueOfCow(Cow x) {
    cout << "The value is " << x << endl;
}
void printValueOfString(string x) {
    cout << "The value is " << x << endl;
}
void printValueOfBarn(Barn x) {
    cout << "The value is " << x << endl;
}
```

Generalizing classes

Where do the parts of a templated class get written? Why?

Class exercise

Convert `Barn` to a templated class

Class exercise

Compiler errors and templated code

```
template <typename T>
void templatedFunction(T& x) {
  if (x < 5) {
    cout << "input is fancy: " << x << endl;
  } else {
    const T y = x;
    y.getCow().danceTheFunkyChicken();
  }
}
```

using_barn.cpp

```
#include <iostream>
#include <cstdlib>
#include <string>
#include "barn.hpp"
using std::cout, std::endl;

int main()  {
    Barn b;
    b.addCow("bessie");
    b.addCow("mabel");

    Barn c;
    c.addCow("mabel");
    c.addCow("bessie");

    if (b == c) {
        cout << "Uh-oh! These should not be equal." << endl;
    }

    for (Barn::iterator i = b.begin(); i != b.end(); ++i) {
        cout << *i << endl;
    }

    Barn::iterator i = b.findCow("bessie");
    if (i != b.end()) {
        cout << "Found " << i->getName() << " in the barn!" << endl;
    }
}
```

cow.hpp

```
#ifndef COW_HPP_INCLUDED
#define COW_HPP_INCLUDED

#include <iostream>
#include <string>

class Cow {
 public:
    Cow() = default;
    ~Cow() = default;
    explicit Cow(const std::string& cowName);

    std::string getName() const;
    bool operator==(const Cow& other) const;
    bool operator!=(const Cow& other) const;
    friend std::ostream& operator<<(std::ostream& output, const Cow& p);
 private:
    std::string name_;
```

```
};
#endif  // cow_hpp_included
```

barn.hpp

```
#ifndef BARN_HPP_INCLUDED
#define BARN_HPP_INCLUDED

#include <string>
#include "cow.hpp"

class Barn {
 private:
    class Iterator;
 public:
    using iterator = Iterator;

    Barn();
    Barn(const Barn& otherBarn) = delete;
    Barn& operator=(const Barn&  otherBarn) = delete;
    ~Barn();
    bool operator==(const Barn& other) const;
    bool operator!=(const Barn& other) const;
    iterator begin() const;
    iterator end() const;
    iterator addCow(const std::string& cowName);
    iterator findCow(const std::string& cowName) const;
 private:
    Cow** cows_;
    size_t size_;
    size_t capacity_;

    class Iterator {
     public:
        using value_type = Cow;
        using reference = value_type&;
        using pointer = value_type*;
        using difference_type = ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;

        Iterator() = default;
        Iterator(const Iterator& other) = default;
        ~Iterator() = default;
        Iterator& operator=(const Iterator& other) = default;
        Iterator& operator++();
        reference operator*() const;
        bool operator==(const Iterator& other) const;
        bool operator!=(const Iterator& other) const;
        pointer operator->() const;
     private:
        friend class Barn;
        Cow** here_;
        explicit Iterator(Cow** here);
    };
};
#endif  // BARN_HPP_INCLUDED
```

barn.cpp

```
#include <string>
#include "barn.hpp"
#include "cow.hpp"

using std::string;

Barn::Barn() : cows_{new Cow*[4]}, size_{0}, capacity_{4}  {
    // nothing (else) to do
}

Barn::~Barn() {
    for (size_t i = 0; i < size_; ++i) {
        delete cows_[i];
    }
    delete[] cows_;
}

Barn::iterator Barn::addCow(const string& cowName) {
    if (size_ == capacity_) {
        capacity_ *= 2;
        Cow** oldcows = cows_;
        cows_ = new Cow*[capacity_];
        for (size_t i=0; i < size_; ++i) {
            cows_[i] = oldcows[i];
        }
        delete[] oldcows;
    }
    cows_[size_] = new Cow{cowName};
    ++size_;
    return Iterator{cows_ + size_-1};
}

Barn::iterator Barn::findCow(const string& cowName) const {
    for (iterator i = begin(); i != end(); ++i) {
        if (i->getName() == cowName) {
            return i;
        }
    }
    return end();
}

bool Barn::operator==(const Barn& other) const {
    if (size_ != other.size_) {
        return false;
    }

    Iterator j = other.begin();
    for (Iterator i=begin(); i != end(); ++i) {
        if (*i != *j) {
            return false;
        }
```

```
        ++j;
    }
    return true;
}

bool Barn::operator!=(const Barn& other) const {
    return !(operator==(other));
}

Barn::iterator Barn::begin() const {
    return Iterator{cows_};
}

Barn::iterator Barn::end() const {
    return Iterator{cows_ + size_};
}

Barn::Iterator::Iterator(Cow** here)
  : here_{here} { }

Barn::Iterator& Barn::Iterator::operator++() {
    ++here_;
    return *this; }

Barn::iterator::reference Barn::Iterator::operator*() const {
    return **here_;
}

bool Barn::Iterator::operator==(const Iterator& other) const {
    return here_ == other.here_;
}

bool Barn::Iterator::operator!=(const Iterator& other) const {
    return !(operator==(other));
}

Barn::iterator::pointer Barn::Iterator::operator->() const {
    return *here_;
}
```

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
- Apply strategies to reduce code complexity and amount of coding ("laziness")
- Convert code to use idiomatic looping constructs
- Specify key design decisions and implementation ideas using pseudocode

C++

C++ Memory Model

You should be able to

- Contrast member initialization against assignment
- Describe and appropriately use `.` and `->`
- 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**
- Explain the benefits and risks of aliasing via pointers
- Describe and detect the following coding errors
  - Double deletion
  - Memory leaks
  - Dangling pointers
  - Null-pointer dereferences
  - Pointer-to-object/pointer-to-array-of-object confusion
- Describe and use the `&`, `*`, and `[]` operators
- 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
- Contrast primitive arrays with the STL's `vector` type

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
- Describe and employ overloading
    - With operators implemented inside the class
    - With operators implemented outside the class
- Describe `friend`ship, and determine when `friend`ship is appropriate
- Determine and describe when and where `const` should be used
- Resolve problems that may occur when `const` is used
- Determine when member functions and data should be declared `static`
- Define and implement iterators
- Specify iterator invalidation semantics, and explain and abide by the iterator invalidation rules for standard STL types

Designing Classes

You should be able to

- Describe and employ encapsulation
- Determine which operations should be placed in the public interface
- Specify the behavior of a class from a user's (interface) perspective
- List and contrast strategies for handling errors
- Employ nested classes

Computational Complexity

You should be able to

- Determine the statement(s) executed most in a code fragment
- Informally analyze code to determine its asymptotic behavior
- Determine and express the performance of code involving loops using $\sum$-notation
- Apply transformations to make code easier to analyze
- Improve algorithms to remove obvious inefficiencies
- Distinguish between worst-case and expected

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 assembly-code file from a single source file
  - An executable program from multiple-object code files
  - A list of the files a source file depends on
- Explain and create Makefiles that include
  - Necessary and sufficient description(s) of file dependencies
  - Standard macro names (e.g, CXX)
  - Standard targets
- Describe and apply the algorithm used by make to rebuild files based on dependencies

Debugging

You should be able to

- Describe and employ strategies for reducing the amount and difficulty of debugging work
- Develop testing strategies
- Employ affirm statements to catch errors

Data Structures

Arrays

You should be able to

- Describe the performance properties of arrays
- Determine whether an array is an appropriate data structure for a given problem
- Describe, implement, & analyze a dynamic array

General Lists

You should be able to

- Suggest operations for generalized singly-linked lists
- Implement (including an iterator that can traverse the structure)
  - Singly-linked lists

General Trees

You should be able to

- Define and explain the following tree concepts:
  - Height
  - Depth
  - Ancestors
  - Descendants
  - Path length
  - Pre-order, post-order, in-order, and level-order traversals
  - Perfect binary trees
- Represent an arbitrary n-ary tree using a binary tree

Binary Search Trees

You should be able to

- Describe the order condition for a BST
- Describe and implement insertion and deletion in a BST
- Describe and implement finding the min and max values in a BST
- Explain the rationale for balancing BSTs, including when doing so is unnecessary
- Explain, apply and implement left and right rotations
- Implement root insetion in a binary tree
- Provide and explain high-level pseudocode for
  - Randomized binary search trees
  - 2-3-4 trees
- Describe the implementation issues that arise in at least one of these approaches
- Contrast the trade-offs and performance differences of each of the above approaches
- Explain the parallels between Red–Black trees 2-3-4 trees

Asymptotic Complexity

For each of the code blocks below, classify T, the number of steps required for input n (and m, where applicable), using asymptotic analysis.

```
for (int i = 0;  i < n;  ++i)
   ++sum;
```

$T(n) \in$ ____

```
for (int i = 0;  i < 2*n;  ++i)
   for (int j = 0;  j < m+1;  ++j)
      ++sum;
```

$T(n, m) \in$ ____

```
for (int i = 0;  i < n;  ++i)
   for (int j = 0;  j < i+1;  ++j)
      ++sum;
```

$T(n) \in$ ____

```
for (int i = 0;  i < n;  ++i)
   ++sum;

for (int j = 0;  j < m;  ++j)
   ++sum;
```

$T(n, m) \in$ ____

```
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < i; j += 2) {
     a[j+1] += 1;
     if (a[j+1] % 2 == 0) a[j] = 2*a[j];
   }
}
```

$T(n) \in$ ____

Analyzing Linked Lists

Consider the `IntList` class that you implemented for Homework 5.

Using the `iterator` you implemented, we could write a new member function `exists` for the `IntList` class:

```
bool IntList::exists(int key) {
    for (iterator i = begin(); i != end(); ++i) {
        if (*i == key) {
            return true;
        }
    return false;
    }

}
```

The following questions refer to an `IntList` that was set up like this:

```
IntList myList;

for (int i=1; i <= N; ++i) {
    myList.push_back(i);
}
```

As a cost metric, we use the number of times we compare a value in our list to the `key` we're looking for.

What will be the cost of calling `myList.exists(1)`? ______

What will be the cost of calling `myList.exists(2)`? ______

What will be the cost of calling `myList.exists(N)`? ______

What would be the *total* cost of looking up *each* of the values in `myList`? ______

What would be the *average* cost of looking up *one* of the values in `myList`? ______

For a list with N elements in it, we can say that `exists` is:

- **Worst case**, $\in \Theta($ ______
- **Average case**, $\in \Theta($ ______

Explain why using the definition of `exists` given above for an `IntList` would be *correct*, but not sufficiently *efficient*, for your `TreeStringSet` class. ______

______

______

Binary Search

If we store values in a **sorted array**, then we can use binary search to look for the value we want:

```
for an array with N elements...

set lo = 0
set hi = N-1

while lo <= hi:
    set mid = (lo + hi + 1)/2
    if (mid == key) return true
    elif key < mid:
        set hi = mid - 1
    else:
        set lo = mid + 1
return false
```

This process is visualized for a successful search for 23 in the image below:

For the array in the list above, which element would have the *smallest* cost in terms of number of comparisons? ____________________

For the array in the list above, which element(s) would have the *largest* cost in terms of number of comparisons? ____________________

For (one of) your answer(s) in the previous question, how many elements in the array would we have to compare against to find the value we wanted? ____________________

Let T(N) represents the number of comparisons needed in the worst case to find an element in an array of N elements.

What is T(1)? ____________________

If we knew T(N/2), how would that help us find T(N)? ____________________

If we knew T(N/4), how would that help us find T(N)? ____________________

Suppose that $N = 2^k$, and solve for T(N). ____________________