Exam 2 Review Questions.
Created by Kyle Parker, Spring 2025

1. Member functions can access a pointer to themselves called what? What is the purpose of this pointer, and can you provide a use case for it?

Solution: Member function can access a self-pointed called `*this*`. The primary purpose of the pointer is to refer to the current object. It is particularly useful when a method parameter shows an attribute with the same name. Further, it is a hidden argument in every (non-static) function calls.

2. What data structure would be best suited for implementing a reverse operation easily?

Solution: A *stack* would be most ideal for a reverse operation. Stacks follow a Last-In-First-Out (LIFO) principle, meaning that the last element added to the stack is the first one to be removed. This behavior makes it simple to reverse the order of elements: as you push elements onto the stack and later pop them off, their order is naturally reversed.

3. We want to create an application like MS Word or Google Docs. What data structure should we utilize for an undo/redo operation?

Solution: Once again, a *stack* would be best as the order is naturally reversed. You want to undo or redo the most recent change made.

4. Suppose we have a attribute defined as: `std::fstream myStream`. Write code for the following prompts (1-4 lines).
    a. Open a file named "myData.dat" for reading.

Solution:
Note: You must use std::ios::in since myStream is of type fstream.
```
myStream.open("myData.dat", std::ios::in);
```

    b. Open a file named "server-dump.log" for writing.

Solution:
```
myStream.open("server-dump.log", std::ios::out);
```

c. Close the file.

Solution:

```
myStream.close()
```

d. Verify the file was successfully opened.

Solution V1:

```
if (myStream.is_open()) {
    // Perform ops here
    // .close must be called within this block:
   myStream.close()
}
```

Solution V2:

```
if (!myStream.is_open()) {
    return; // Return an error value as necessary.
}
```

5. Define abstraction, one of the four pillars in object-oriented programming.

Solution: In C++, abstraction

6. Define encapsulation, one of the four pillars in object-oriented programming.

Solution: Encapsulation is the idea of binding functions and attributes. This enables the ability to restrict access to data (i.e., some variables can be made read-only). Further, if there are any attributes required behind the scenes (i.e., an fstream attribute), these can be hidden.

7. Define the following data structures: queue, stack, binary search tree (BST), and singly linked list. For each data structure, provide specific examples of real-world applications.

Solution:

1. Queue
    a. Linear data structure which processes elements in the order they were received. Follows First-In-First-Out (FIFO) principle. We enqueue at the **back** and dequeue from the **front**.

   b. Think about a call center, networking operations, or printer job. The first person/operation will always be processed first. However, if we have a priority queue, this becomes a tree and is not the same as a queue.
2. Stack
   a. Linear data structure which processes elements based on the most recently received. Follows First-In-Last-Out (FILO) principle. We push and pop from the front.
   b. In addition to questions 2 and 3, it is used for function calls and plates in a cafeteria.
3. Binary Search Tree (BST)
   a. **Non-linear** hierarchical data structure. We are guaranteed everything to the left of any given element is less than that node. Similarly, all elements on the right are greater than that node. The cluster of elements/nodes on the right or left of a particular node can be referred to as a subtree.
   b. Some database systems, sorted data, etc.
4. Singly linked list
   a. Linear data structure. A collection of nodes which are connected by a pointer. All linear data structures can utilize a singly linked list as the underlying data structure. There are many other data structures which can be built using a singly linked list. Only sequential access is supported (we cannot jump to the 5th element; we must visit 1…4 first.)
   b. Streaming playlists (we will never need to play a previous song). Underlying data structure for many linear data structures. Cannot be used on non-linear data structure as there is a single pointer.

8. Write a recursive `insert(const std::string& newData)` member function for the BST class. **The function should return true if the new data was successfully inserted and false otherwise.** Assume that the BST uses dynamically linked nodes, where each node contains a string data member and pointers to its left and right children. You may define a private helper function with different parameters to facilitate the recursion; however, the public version must remain as specified above.

   Important note: All variables preceded with an underscore (_) denote a private variable; assume getters exist for Node.
   Class BST has `Node * _root`.
   Class Node has `Node* _left`, `Node* _right`, `string _data`.

Note on all solutions: I chose to ignore duplicate values. Ensure your solution does not insert duplicates. Print statements are acceptable.

Solution V1:

// Assumption for V1: Node's getLeft() returns a **reference** to the pointer

```
// Public
bool insert(const std::string& newData) {
    return insert(newData, _root);
}

// Private
bool insert(const std::string& newData, Node*& pTree) {
    bool success = false;
    if (pTree == nullptr) {
         success = true;
         pTree = new Node(newData);
    }
    else if (newData < pTree->getData()) {
        success = insert(newData, pTree->getLeft());
    }
    else if (newData > pTree->getData()) {
        success = insert(newData, pTree->getRight());
   }

    return success;
}
```

Solution V2:

```
// Public
bool insert(const std::string& newData) {
    return insert(newData, _root);
}

// Private
bool insert(const std::string& newData, Node*& pTree) {
    bool success = false;
    if (pTree == nullptr) {
         pTree = new Node(newData);
         success = true;
    }


// If statement continued on next page.
```

```
    else if (newData < pTree->getData()) {

       if (pTree->getLeft() != nullptr) {
          success = insert(newData, pTree->getLeft());
       } else {
          pTree->setLeft(new Node(newData);
          success = true;
       }

    }
    else if (newData > pTree->getData()) {

      if (pTree->getRight() != nullptr) {
          success = insert(newData, pTree->getRight());
       } else {
          pTree->setRight(new Node(newData);
          success = true;
       }

    }

    return success;
}
```

9. Write a recursive `destroyTree()` member function for the BST class which reclusively destroys the current tree. This function should *recursively* deallocate all nodes in the current tree to free up memory. You may define a private helper function with different parameters to facilitate the recursion.

   Important note: All variables preceded with an underscore (_) denote a private variable; assume getters exist for Node.
   Class BST has `Node * _root`.
   Class Node has `Node* _left`, `Node* _right`, `string _data`.

Solution:
// Public:
// Either a public function or BST destructor is accepted. This question called for a public version of destroyTree. In practice, it is rare we would expose such a function; most of the time, it will only be called from the BST's destructor.

```
void destroyTree() { // Or you can use ~BST() {
    destroyTree(_root);
}

// Private
void destroyTree(Node*& node) {
    if (node != nullptr) {
        destroyTree(node->getLeft());
        destroyTree(node->getRight());
        delete node;

        node = nullptr; // Not required for all implementations
    }
}
```

10. Create a structure which will hold a GPS point, title, an annotation, and a unique id. Here are the data types:
    GPS point: CLLocation
    Title: std::string
    Annotation: std::string ***CONSTRAINT: Must be less than 20 chars***
    Unique ID: UUID

    Create declarations of an explicit constructor and required attributes. We want all attributes to be non-mutable.
    Do not use default values. Assume required include statements are present.

You have two options to make the attributes non-mutable. The way we have done in class is shown below in the private section (getters omitted.) Another way, by using const is shown in the public section. There are pros and cons to both approaches.

```
struct PinPoint {
        public: // All operations should be public.

// Constructor
PinPoint(CLLocation location,
         std::string title,
         std::string annotation,
         UUID uuid);

// Alternative, const approach
const CLLocation location;
const std::string title;
const std::string annotation;
const UUID uuid;

        private: // All attributes should be private.

// Typical approach in this class.
CLLocation _location;
std::string _title,
            _annotation;
UUID _uuid;


}
```

// Operator is below.

// Place your declaration for an output stream operator. Display the data in JSON format. Assume the output stream operator is already overloaded for UUID and CLLocation. Sample:

```
{
  uuid: 5B3D1A2E-3F4C-4B5D-8A6E-7B8C9D0E1F2A,
  title: "My pinpoint",
  annotation: "My pinpoint's annotation",
  point: { lat: 38.8409, long: - 105.0422 }
}
```

Solution:

// The const and reference on PinPoint are not required. The names can also vary.

```
ostream& operator<<(ostream& lhs, const PinPoint& rhs);
```

a. Define the constructor declared for PinPoint.

Solution:

// This works for both the const and non-const approach. However, if you used const, you must use the initializer list shown. Any const member must be initialized using an initializer list. If you would like more information and pros/cons of using a const member, investigate it!

// Solution V1: const approach

```
PinPoint(CLLocation location,
         std::string title,
         std::string annotation,
         UUID uuid)
     : location(location),
       title(title),
       annotation(annotation),
       uuid(uuid) {}
```

// Solution V2: traditional approach

```
PinPoint(CLLocation location,
         std::string title,
         std::string annotation,
         UUID uuid)
     : _location(location),
       _title(title),
       _annotation(annotation),
       _uuid(uuid) {}
```

b. Define the output stream operator. **You CANNOT use the friend operator. Write down any assumptions made.**

// Assumptions for V1 and V2:
// 1. Using statements for std::ostream and std::endl are declared.

Solution V1: const approach

{
 uuid: 5B3D1A2E-3F4C-4B5D-8A6E-7B8C9D0E1F2A,
 title: "My pinpoint",
 annotation: "My pinpoint's annotation",
 point: { lat: 38.8409, long: - 105.0422 }
}

```
ostream& operator<<(ostream& lhs, const PinPoint& rhs) {
   lhs << "{" << endl;

   lhs << "\tuuid: " << rhs.uuid << ',' << std::endl;
   lhs << "\ttitle: " << rhs.title << ',' << std::endl;
   lhs << "\tannotation: " << rhs.annotation << ',' <<
std::endl;
   lhs << "\tpoint: " << rhs.location << std::endl;

   lhs << "}";
   return lhs; // Do not forget this line!
}
```

Solution V2: traditional approach

// Assumption: getters are defined

```
ostream& operator<<(ostream& lhs, const PinPoint& rhs) {
   lhs << "{" << endl;

   lhs << "\tuuid: " << rhs.getUUID() << ',' << std::endl;
   lhs << "\ttitle: " << rhs.getTitle() << ',' << std::endl;
   lhs << "\tannotation: " << rhs.getAnnotation() << ',' <<
std::endl;
   lhs << "\tpoint: " << rhs.getLocation() << std::endl;

   lhs << "}";
   return lhs; // Do not forget this line!
}
```

11. Write a function which accepts a queue and sum all values within. You **only** have access to `enqueue`, `dequeue`, and `isEmpty`. Ensure you restore the queue to its original state (that is it holds the same values in the same order).

Solution:

// Assumption: dequeue returns the value removed.

```
// Note: It is ok if you do not use a template.
int sumQueueValues(Queue<int>& q) {
     Queue<int> tempQ;
     int sum = 0;
     int temp = 0;

    while (q.isEmpty() == false) {
         temp = q.dequeue();
         sum += temp;
         tempQ.enqueue(temp);
    }

    while (tempQ.isEmpty() == false) {
        q.enqueue(tempQ.dequeue());
    }

    return sum;
}
```

```
// Full template version
template <typename T>
T sumQueueValues(Queue<T>& q) {
     Queue<T> tempQ;
     T sum = 0;
     T temp = 0;

    while (q.isEmpty() == false) {
         temp = q.dequeue();
         sum += temp;
         tempQ.enqueue(temp);
    }

    while (tempQ.isEmpty() == false) {
        q.enqueue(tempQ.dequeue());
    }

    return sum;
}
```

12. Write a function which accepts a std::string and converts it to an integer. This should work *similar* to stoi. However, only 10 digits are permitted (assume max value is 9,999,999,999.) It is valid to have a preceding + to indicate a positive value. Otherwise, only digits are permitted. Further, negative values are considered illegal. Return -1 for invalid inputs.

    Samples:
    Input: "0987654356789765435678 9" → Result: -1
    Input: "12452" → Result: 12452
    Input: "+24" → Result: 24
    Input: "2i9" → Result: -1
    Input: "-24" → Result: -1

Solution V1: for-each

// Assumption: <cmath> is included

// Assumption: <stack> is included

// Assumption: using statements defined as needed

```
int mystoi(const std::string& str) {
   int sum = 0;
   int current = 0;
   int power = 1;

   stack<char> s;


   for (char& c : str) {
      s.push(c);
   }

   while (s.isEmpty() == false) {

      char c = s.top();
      s.pop();

      if (c == '+' && power == 1) {
         // Nothing to do
      } else {
         current = c - '0';

         // Something is not a digit.    Or num is too large
         if (current < 0 || current > 9 || power >= 10) {
            return -1;
         }

         sum += power * current);
         power *= 10;
      }

   }

   return sum;
}
```

Solution V2:

// Assumption: <cmath> is included.

// Assumption: <cctype> is included.

```
int mystoi(const std::string& str) {
    int sum = 0;
    int current = 0;
    int power = 1;

    bool valid = !str.isempty() && (str[0] == '+' ||
                                    isdigit(std[0]));

    for (int i = str.size() - 1; i >= 0 && valid; --i) {
        current = str[i] - '0';

        if ((i != 0 && (current < 0 || current > 9)) ||
            (i == 0 && str[0] != '+')
           ) { // Invalid character
            valid = false;
        }
        else if (power >= 10) { // Too large of number
            valid = false;
        }
        else { // Everything is valid!
            sum += power * current;
            power *= 10;
        }
    }

    // If valid is true, return the sum, otherwise return -1.
    return valid ? sum : -1;
}
```

13. Answer the following questions about this BST.
    a. Is this a proper BST?

Solution: Yes, this a proper BST. At any given element, the value to the left is always less and the right is greater.

    b. What is a valid sequence of insertions that would result in the same binary search tree (BST) structure?

Solutions:

1. 10 5 15 3 7 12 18 1 4 6 8 14
2. 10 15 5 12 3 19 7 6 14 4 8 1
3. 10 5 15 7 18 3 12 14 4 1 6 8
4. Any solution where all numbers on a given level are inserted prior the level below. That is, the following groups are inserted in order. Note: A level in a tree is one which the node has the same height. e.g., 5 and 15 are at the same height, so they are on the same level.
    a. 10
    b. 5 15
    c. 3 7 12 18
    d. 1 4 6 8 14

    c. What is the height of this tree?

Solution:
The height is determined by the longest path from the root node to any leaf node. In this case, counting each edge from node 10 to node 1 shows there are 3 edges in total, so the height is 3. (An edge is the line connecting two nodes.)

    d. What is the depth of this tree?

Solution:
The depth of this tree is determined by the longest path from any leaf node to the root. In this example, the longest path goes from node 14 to node 10 and passes through 3 edges. Therefore, the depth is 3.

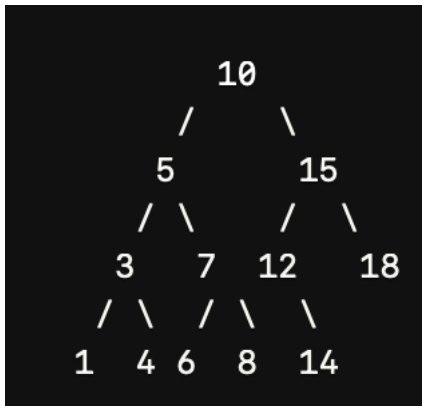
Note: For the overall tree (the root), height and depth are the same. However, for individual nodes, the depth (distance from the root) may differ from their height (distance to the deepest leaf).

e. What is the height of node 12?

Solution:
The height of node 12 is 2 since we pass two edges from the root.

f. What is the depth of node 5?

Solution:
The depth of node 5 is 2 since we pass two edges from the furthest leaf.



14. **True/False**: It is possible to change the precedence of the addition operator (+) to be higher than that of the multiplication operator (*).

Solution: **FALSE** It is never possible to change the precedence or overload an operator that does not already exist.

15. **True/False**: A template function always has identical behavior to overloaded functions.

Solution: **FALSE** There are times where the behavior may appear identical, but they have different purposes and are not interchangeable. Templates are to be used with any type whereas overloaded functions define explicit types. However, we can mock the behavior of overloaded functions to an extent (see question 16).

16. Write a template function which accepts two parameters to mock the behavior of an overloaded function, add. Assume the first parameter type is the return type as shown below with overloaded operators.
    double add(double a, int b)
    int add(int a, int b)
    float add(float a, int b)

Solution:

/// Precondition: a and b must have an overloaded operator + that handles the addition of type U on type T. (that is `T + U` is defined)

```
template <typename T, typename U>
T add(T a, U b) {
    return a + b
}
```