Today

- Quiz 2
- Exceptions
- Copy and Move

Assignments

- HW 2 out
Exceptions in C++

Exceptions provide a separate “flow of control” in a program

- typically for handling runtime errors more “gracefully” (graceful degradation)
- exceptions are **thrown** and **caught**
- values typically can be thrown to provide information about the exception
- We’ll use exceptions for handling out-of-bound indexes

Example:

```cpp
// throwing within a function ...

```
```
template <typename T>
void LinkedSeq<T>::insert(const T& elem, int index)
{
    // check for valid index
    if (index < 0 || index > size())
        throw std::out_of_range("LinkedSeq<T>::insert(const T&, int)");
    ...
}
```

```cpp
// catching as part of calling a function ...

```
```
void some_other_function()
{
    LinkedSeq<char> s;

    s.insert('a', 1); // throws an uncaught exception

    try {
        s.insert('a', 1); // throws a caught exception
    } catch (std::out_of_range& ex) {
        // do something to handle error
        cout << "out of range exception: " << ex.what() << endl;
    }
}
```
C++ (unlike Java) does not require handling (catching) of exceptions

- e.g., first insert results in a runtime error:

```
terminate called after throwing an instance of 'std::out_of_range'
  what():  LinkedSeq<T>::insert(const T&, int)
Aborted (core dumped)
```

In the example, the caught exception prints:

```
out of range exception: LinkedSeq<T>::insert(const T&, int)
```

Note C++ has many exception classes for various errors

- of which, `std::out_of_range` is just one of many
Essential Operations

Operations we have to consider when defining classes:

- construction ...
  e.g., `LinkedSeq<int> s1;`
- initialization ...
  e.g., `LinkedSeq<int> s2 = s1;`
- copy and assignment ...
  e.g., `tmp = s2;`
- move ...
  more later
- destruction ...
  out of scope or `delete`

Constructors, destructors, and copy and move operations are logically interwined

- if a non-trivial destructor is required, then typically need to define all
- e.g., classes that require heap/dynamic allocation

The full complement of the (essential) functions for a class `X`:

```cpp
class X
{
public:
  X(some params);    // overloaded constructor
  X();              // default constructor
  X(const X& x);    // copy constructor
  X(X&& x);         // move constructor
  X& operator=(const X& x); // copy assignment: clean up target and copy
  X& operator=(X&& x); // move assignment: clean up target and move
  ~X();             // destructor: clean up
...}
```
Compiler provides default implementations of each (not overloaded constructor)

- for construction, do nothing
- for clean up, do nothing
- for copy, just copy member variables

... “shallow” vs “deep” copy

Q: For our LinkedSeq class, what could go wrong?

(1) memory leaks

... filling heap with unreachable memory

- when object goes out of scope ...
- linked list nodes stay in heap

(2) memory “entanglement”

... two objects share same structure

```
LinkedSeq<char> s1;
s1.insert('a', 0);
s1.insert('b', 1);
LinkedSeq<char> s2(s1);  // default copy constructor
```
s2.insert('c', 2);
s1.contains('c'); // evaluates to true!

Note that the behavior depends on how functions implemented ...

- e.g., if loop until reach tail, versus until we reach `nullptr`

Q: What would happen if a destructor were implemented and called on s1?

- s2’s head pointer would point to a deleted node
- and same for s2’s tail pointer (depending on implementation)
(3) “Entanglement” and memory leaks with default copy assignment:

```cpp
LinkedSeq<char> s1;
s1.insert('a', 0);
s1.insert('b', 1);

LinkedSeq<char> s1;
s2.insert('c', 0);

s1 = s2;       // default copy assignment
```

![Diagram showing entanglement and memory leak with default copy assignment](image)

- **S1**: `a` → `b` → `null`
- **S2**: `c` → `null`
- After default copy assignment, `s1` refers to the same memory as `s2`, potentially causing a memory leak if not properly managed.
Hints for implementing copy functions:

Q: Why the return type in an assignment operator? (i.e., explain the signature)

```cpp
template<typename T>
LinkedSeq<T>& LinkedSeq<T>::operator=(const LinkedSeq<T>& rhs)
{
    ...
}
```

Allows = to be chained together ... e.g., sometimes you’ll see:

```cpp
while ((p = p->next) != nullptr)
{
    ...
}
```

Q: What should happen when we do the following?

```cpp
LinkedSeq<char> s1;
...
```

```cpp
s1 = s1; // bad things happen here if not careful!
```

Q: How do we prevent bad things from happening in this case?

```cpp
template<typename T>
LinkedSeq<T>& LinkedSeq<T>::operator=(const LinkedSeq<T>& rhs)
{
    if (this != &rhs)
    {
        // delete lhs (current object) nodes ... see make_empty()
        // `deep' copy rhs nodes to lhs one node at a time
        // update node_count for lhs
    }
    return *this;
}
```
Note we can reuse copy assignment in copy constructor:

```cpp
template <typename T>
LinkedSeq<T>::LinkedSeq(const LinkedSeq<T>& rhs) {
    *this = rhs;  // calls the assignment operator!
}
```

Q: Do we need to do anything special for this to work?

- no ...
- since *this is just a 0-length linked list
The move operations

Move helps avoid excessive copies ...

```cpp
LinkedSeq<int> add_one(const LinkedSeq<int>& s)
{
    LinkedSeq<int> tmp;
    for(int i = 0; i < s.size(); ++i)
        tmp.insert(s[i] + 1, i);
    return tmp; // Q: what happens here?
}

int main()
{
    LinkedSeq<int> s1;
    s1.insert(1, 0);
    // ...
    LinkedSeq<int> s2 = add_one(s1); // Q: what happens here?
    cout << s2 << endl;
}
```

The compiler will try to reduce the copying by using move instead

- but, default move ops won’t be created if other essential operators defined
- so we have to define our own ...

Move is much simpler than copy ...

(1) we first “transfer” the data structure to the target (lhs)

(2) and then “zero-out” the associated variables in the source (rhs)
Move basics

- \(X&\) is (now) called an "\textit{lvalue reference}\" ... can be assigned to
- \(X&&\) is called an "\textit{rvalue reference}\" ... is assigned from
- to force a move, can use the \texttt{std::move(T&& t)} function ... see unit tests

The move constructor can simply call the move assignment using \texttt{move} ...

\begin{verbatim}
    template<typename T>
    LinkedSeq<T>::LinkedSeq(LinkedSeq<T>&& rhs)
    {
        // defer to move assignment
        // move() just returns an rvalue reference to rhs
        *this = std::move(rhs);
    }
\end{verbatim}

The move assignment for our linked list implementation ...

\begin{verbatim}
    template<typename T>
    LinkedSeq<T>& LinkedSeq<T>::operator=(LinkedSeq<T>&& rhs)
    {
        if (this != &rhs) { // Q: Why do this?
            // delete lhs ( current object ) nodes ... see make_empty ()
            head = rhs.head; // transfer to lhs
            ...
            rhs.head = nullptr; // zero-out rhs
            ...
        }
        return *this;
    }
\end{verbatim}
Additional hints

The `make_empty()` private helper:

- does the work of deleting the nodes in a linked list
- the destructor and (copy and move) assignments can then reuse the functionality

The two `operator[]` functions:

- both check for valid indexes (if not, throws exception like in insert)
- both involve traversing the linked list
- one returns value of corresponding linked list node (rvalue)
- one returns value reference to be set (lvalue)
The full `LinkedSeq` class:

```cpp
template<typename T>
class LinkedSeq : public Sequence<T>
{
public:

    // essential operations
    LinkedSeq();
    LinkedSeq(const LinkedSeq& rhs);
    LinkedSeq(LinkedSeq&& rhs);
    LinkedSeq& operator=(const LinkedSeq& rhs);
    LinkedSeq& operator=(LinkedSeq&& rhs);
    ~LinkedSeq();

    // sequence operations
    int size() const override;
    bool empty() const override;
    T& operator[](int index) override;
    const T& operator[](int index) const override;
    void insert(const T& elem, int index) override;
    void erase(int index) override;
    bool contains(const T& elem) const override;
    void sort() override;

private:

    // linked list structure
    struct Node {
        T value;
        Node* next = nullptr;
    };

    // member variables
    Node* head = nullptr;
    Node* tail = nullptr;
    int node_count = 0;

    // helper function
    void make_empty();
};
```