Today

• C++ Essential Operations

Assignments

• HW-2 due fri
• Exercises …
• Quiz-3 thur
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 and deallocation

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

```cpp
LinkedSeq<char> s1;
s1.insert('a', 0);
s1.insert('b', 1);
LinkedSeq<char> s2(s1); // default copy constructor
```

![Diagram](attachment:image.png)
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:

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

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

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

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

```
s2: head tail
     'c' →
```

```
s1: head tail
     'a' | 'b' | →
```

```
s2: head tail
     'c' →
          ←→
```

```
s1: head tail
     'a' | 'b' | ←→
          ←→
```

```
s2: head tail
     'c' →
          ←→
```

```
s1: head tail
     'a' | 'b' | ←→
          ←→
```

```
s2: head tail
     'c' →
          ←→
```
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;
...
sl = 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) {
        // clear lhs (current object)
        // 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 ...
- *this is just a 0-length linked list
- we initialize the member variables in declarations
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) ... like “stealing”
2. and then “zero-out” the associated variables in the source (rhs)
Move basics

- \( \texttt{X\&} \) is (now) called an “\textit{lvalue reference}” ... can be assigned to
- \( \texttt{X\&\&} \) 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?
        // clear() lhs list
        head = rhs.head; // transfer to lhs
        ...
        rhs.head = nullptr; // zero-out rhs
        ...
    }
    return *this;
}
\end{verbatim}
Additional hints

Can reuse the `clear()` function:

- does the work of deleting the nodes in a linked list
- destructor and (copy and move) assignments can 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)
- both have same implementation (just different signatures!)

Optimize the “end” cases with the `tail` pointer:

- inserting at end of sequence (e.g., `s.insert('a', 4)` if `s.size()==4`)
- accessing / updating last sequence element
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;
    void clear() 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;
};
```