Today

- Move
- Resizable Arrays (intro)

Assignments

- HW 2 out
- Quiz 3 (dynamic binding, essential ops)
**Essential Operations (Review)**

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
...  
};
```
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 "lvalue reference" ... can be assigned to
- $X&&$ is called an "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}
The ArraySeq class

For HW-3, you will implement a “resizable array” (called ArraySeq)
• Similar to Java’s ArrayList and C++’s Vector classes

Resizable arrays address fixed-sized issue of normal arrays
• Arrays in C++ are a fixed size at creation
• Once we reach the capacity of the array, can’t add more elements

Basic idea of a resizable array:
• start with an initial capacity (in our case, 0 slots)
• once all slots full, next add dynamically “resizes” the array

Resizing the array involves four steps:
(1). create a new array with twice the current array’s capacity (e.g., 1, 2, 4, 8, ...)
(2). copies all current elements to the new array
(3). deletes the current array
(4). sets the current array to the new array

Resizable arrays introduce a “trade off” in terms of efficiency
• Solves the fixed-size problem
• Maintains fast access to elements
• (Slightly) increases the cost of adding (because of resizing)