Today

- Quiz 2
- C++ Odds and Ends for HW-2

Assignments

- EX-3
- HW-2 out
A Basic “Sequence” ADT

A bare-bones Sequence class where items are stored as integers

- Note this class has problems we are going to fix as we go ...

```cpp
class Sequence
{
public:
    int size() const; // number of elems
    bool empty() const; // true if empty
    void clear(); // removes all items
    int& operator[](int index); // set value at index
    const int& operator[](int index) const; // get value at index
    void insert(int elem, int index); // grow at index
    void erase(int index); // shrink at index
    bool contains(int elem) const; // check membership
};
```

Examples:

```cpp
Sequence s;
s.insert(10, 0);
s.insert(20, 1);
cout << s[0] << " " << s[1] << endl; // prints: 10 20
s[0] = 15;
s.erase(1);
cout << s.size() << " " << s[0] << endl; // prints: 1 15
```

Q: How should we handle invalid indexes?

- Option 1: do nothing
- Option 2: throw exception (checked indexes) ... our approach, more later
A linked list subclass implementation

A linked-list version as a subclass of Sequence

- *note again there are issues below, which we'll fix as we go!*

```cpp
class LinkedSeq : public Sequence
{
public:
    // constructors, destructors, copy/move operators ...

    // overridden from Sequence
    int size() const;
    bool empty() const;
    void clear();
    int& operator[](int index);
    const int& operator[](int index) const;
    void insert(int elem, int index);
    void erase(int index);
    bool contains(int elem) const;

private:
    struct Node {
        int value;
        Node* next = nullptr;
    };
    Node* head = nullptr;
    Node* tail = nullptr;
    int node_count = 0;
};
```

There are numerous issues with the above implementations ...

Q: Can you spot them?
Problem 1: Static vs Dynamic Binding

Consider the following:

```cpp
LinkedSeq s;
s.insert(42, 0);
```

Q: Which version of `insert` is used (Sequences's or LinkedSeq's)?

- **LinkedSeq::add** is used
- Determined by the compiler at **compile time** ... i.e., "statically"

The **LinkedSeq::insert** function definition is "bound" to the call

- Since it occurs at compile time, it is an example of "static (function) binding"
- The **compiler** determines object `s` is of type **LinkedSeq**

Now consider the following:

```cpp
void insert_in_front(const Sequence& s2, int val)
{
    s2.insert(val, 0);
}
...
LinkedSeq s1;
s1.insert(20, 0);
insert_in_front(s1, 10);
```

An example of "subtype polymorphism" ... using subclass in place of superclass

- **insert_in_front** can take a **Sequence** or any of its subclasses
Another similar example (but without the function):

```cpp
LinkedSeq s1;
s1.insert(20, 0);
...
Sequence& s2 = s1; // using pointers: Sequence* s2 = &s1;
s2.insert(10, 0); // using pointers: s2->insert(10, 0);
...
```

Q: In these examples, which version of `insert` is used (Sequence's or LinkedSeq's)?

- following the same static binding rules ...
- for `s1`, `LinkedSeq::insert` is used ... what we want
- for `s2`, `Sequence::insert` is used! ... not what we want

For this to work as expected, we need C++ to use "dynamic binding"

- use the object's actual type at runtime instead of the compiler's static type
- C++ has to be told explicitly to use dynamic binding (per function)
- which is done using the `virtual` specifier
New version of the `Sequence` class (for dynamic binding):

```c++
class Sequence
{
public:
  virtual ~Sequence() {} // required for subclasses
  virtual int size() const;
  virtual bool empty() const;
  virtual void clear();
  virtual int & operator[](int index);
  virtual const int & operator[](int index) const;
  virtual void insert(int elem, int index);
  virtual void erase(int index);
  virtual bool contains(int elem) const;
};
```

Notes on `virtual` functions:

- subclass functions overriding virtual functions are virtual (can leave off `virtual`)
- can use `override` specifier for subclass functions (compiler check)

```c++
void insert(int elem, int index) override;
```
Problem 2: Sequence shouldn’t prescribe an implementation

As an ADT, our Sequence class doesn’t suggest a specific data structure

- there are many possible ways to implement the type ...

Instead, we should make Sequence an “abstract” class

- we do this in C++ by declaring its functions to be “pure virtual”

New version of Sequence as an Abstract Class:

class Sequence
{
public:
    virtual ~Sequence() {} // required for subclasses
    virtual int size() const = 0;
    virtual bool empty() const = 0;
    virtual void clear() = 0;
    virtual int & operator[](int index) = 0;
    virtual const int & operator[](int index) const = 0;
    virtual void insert(int elem, int index) = 0;
    virtual void erase(int index) = 0;
    virtual bool contains(int elem) const = 0;
};

- the “= 0” annotation states we aren’t implementing the function
- a class with one “pure” function is considered “abstract”
- an abstract class can’t be instantiated
- subclasses make some or all functions “concrete” (by implementing them)
**Problem 3: Sequences of any type of value**

Sequences should be able to store values of any type

- otherwise, we'd need classes `IntSequence`, `DoubleSequence`, etc.
- We want our Sequence types to be “generic” (in terms of stored value types)

C++ **templates** allow us to define “type parameters”

- in our example, we replace `int` with the type parameter

```cpp
template<typename T>
class Sequence
{
public:
    virtual int size() const = 0;
    virtual bool empty() const = 0;
    virtual void clear() = 0;
    virtual T& operator[](int index) = 0;
    virtual const T& operator[](int index) const = 0;
    virtual void insert(const T& elem, int index) = 0;
    virtual void erase(int index) = 0;
    virtual bool contains(const T& elem) const = 0;
};
```

To declare a `Sequence` type, we must pass in the type argument:

```cpp
// pointer to sequence of int
Sequence<int>* int_seq_ptr = nullptr;

// pointer to sequence of double
Sequence<double>* double_seq_ptr = nullptr;

// pointer to sequence of string
Sequence<string>* string_seq_ptr = nullptr;

// pointer to a seq of seq of string (pointers)
Sequence<Sequence<string>*>* string_seq_seq_ptr = nullptr;
```
We also use template parameters for subclasses:

```cpp
template<typename T>
class LinkedSeq : public Sequence<T>
{
public:
    // constructors, destructor, assignment ops

    // overridden from Sequence
    int size() const override;
    bool empty() const override;
    void erase() 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;

private:
    struct Node {
        T value;
        Node* next = nullptr;
    };
    Node* head = nullptr;
    Node* tail = nullptr;
    int node_count = 0;
};
```

We provide the concrete types when declaring and instantiating:

```cpp
LinkedSeq<int> int_seq;
LinkedSeq<string> string_seq;
Sequence<double>* seq_ptr = new LinkedSeq<double>;
```
To implement member functions, must include type params

```cpp
template<typename T>
int LinkedSeq<T>::size() const
{
    return node_count;
}

template<typename T>
bool LinkedSeq<T>::empty() const
{
    return size() == 0;
}
...

template<typename T>
void LinkedSeq<T>::insert(const T& elem, int index)
{
    ...
}
```

The template declaration is part of the function signature

- every function has to announce the template type
- annoying but required

Also, because of issues with compilation ... 

- need to include the function implementations in the class header (.h) file
- also annoying but required
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
- we’ll only 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++ does not require you to handle (i.e., “check”) exceptions

The first insert above results in a runtime error (“unchecked” exception):

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

The second insert is “checked”, and so would print:

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

Note C++ has many exception classes for various errors

- `std::out_of_range` is just one of many