Today

- Dynamic Binding
- Abstract Classes
- Templates

Assignments

- HW-2 out
- Quiz 2 on Thursday
Basic Sequence ADT and LinkedList Implementation

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

class Sequence
{
public:
    int size() const; // returns length
    bool empty() const; // true if empty
    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
};

Note again that there are issues below, which we'll fix as we go!

class LinkedSeq : public Sequence
{
public:
    // constructors, destructors, copy/move operators ...
    // overridden from Sequence
    int size() const;
    bool empty() const;
    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;
    };
    Node* head;
    Node* tail;
    int length;
};
**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);
...
```

Exercise 3:

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):

```cpp
class Sequence
{
public:
    virtual int size() const;
    virtual bool empty() const;
    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)

```cpp
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 int size() const = 0;
    virtual bool empty() const = 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)

**Exercise 4 and 5:**

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 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;
    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;
    };
    Node* head;
    Node* tail;
    int length;
};
```

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 length;
}

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