

# 601.220 Intermediate Programming

Summer 2023, Meeting 21 (July 21nd)

# Today's agenda

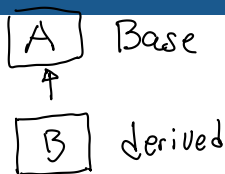
- Review exercise 33
- Day 35 recap questions
- Exercise 35
- Day 36 recap questions
- Exercise 36

## Reminders/Announcements

- Final project due by 11pm on **Friday, July 28th**
- Final exam in class on **Friday, July 28th**
  - Review materials are posted on the course website
  - Final exam review session: next week, we will announce on Piazza when details are finalized
- No office hours on Sunday and Saturday. I will be taking additional office hours by appointment if needed.

## Exercise 33 review

a → private



Add accessor function for member variable a to class A:

```
int get_a() const { return a; } ⇒ defined in A
```

B (and other classes derived from A) will need these to get the values of this member variable. (The member variable d can be accessed directly because it is protected rather than private.)

## Exercise 33 review

A::toString member function:

```
virtual std::string toString() const {  
    std::stringstream ss;  
    ss << "[Aclass: a = " << a  
        << ", d = " << d  
        << ", size = " << sizeof(*this)  
        << "];"  
    return ss.str();  
}
```

## Exercise 33 review

```
A & a = b_inst();  
a.toString();
```

B::toString member function:

```
virtual std::string toString() const override {  
    std::stringstream ss;  
    ss << "[Bclass: a = " << get_a()  
        << ", b = " << b  
        << ", d = " << d  
        << ", size = " << sizeof(*this)  
        << "];"  
    return ss.str();  
}
```

Because the a member variable in the base class A is private, it's necessary to call a getter function to access its value.

## Exercise 33 review

in `main()`, the following statement does not compile:

```
bobj = aobj; → fail
```

With a cast of `bobj` to `A&` (reference to `A`), we can assign to just the `A` part of `bobj`:

```
static_cast<A&>(bobj) = aobj;
```

## Exercise 33 review

fun() pure virtual member function in class A:

```
virtual int fun() const = 0;
```

Implementation in class B:

```
virtual int fun() const override {  
    return int(get_a() * b * d);  
}
```

Note that A is no longer instantiable, so variable definitions like

```
A aobj(10);
```

are no longer allowed.



## Exercise 33 review

```
class C : public A {
private:
    int e;

public:
    C(int val = 0) : e(val) { } // sets a and d to 0 using
                               // A's default ctor
    virtual std::string toString() const override {
        std::stringstream ss;
        ss << "[Cclass: a = " << get_a()
            << ", d = " << d
            << ", e = " << e
            << " size = " << sizeof(*this)
            << "];";
        return ss.str();
    }
}
```

## Exercise 33 review

```
// C class continued
```

```
virtual int fun() const override {  
    return int(get_a() * d * e);  
}  
};
```

## Day 35 recap questions

- 1 What is the difference between an unscoped and a scoped enum?
- 2 Why do we use exceptions?
- 3 What keyword is used to generate an exception? What keyword indicates that the block of code may generate an exception? What keyword indicates what should be done in the case of an exception?
- 4 In the case of multiple matching `catch` blocks for a thrown exception, which one actually catches the exception?
- 5 How do you get the message associated with an exception?

# 1. What is the difference between an unscoped and a scoped enum?

An unscoped `enum` type adds the enum members to the current namespace.

- \* The members of a scoped `enum` type are placed in the namespace of the `enum` type.

## Unscoped vs. scoped enum types

```
enum Color {  
    RED, GREEN, BLUE  
};
```

```
// ...elsewhere in the program...  
Color c = BLUE;
```

```
enum class Color {  
    RED, GREEN, BLUE  
};
```

```
// ...elsewhere in the program...  
Color c = Color::BLUE;
```

Scoped enumerations are generally preferred because they do not “pollute” the namespace they’re in.

## 2. Why do we use exceptions?

Exceptions help us separate

- where in the program error conditions might occur, from
- where in the program it makes sense to handle the error conditions

By using exceptions, we can write functions with the attitude that they will succeed.

If an error condition arises, we can throw an exception.

Exceptions allow us to only handle error conditions in the specific points in the program where we are prepared to deal with them, and not clutter the rest of the program with complicated and hard-to-test error handling paths.

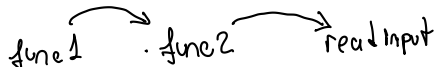
## Error handling without exceptions

```
// Read an integer, then read that many double values  
// and add them to the given vector  
bool read_input(std::istream &in, std::vector<double> &v) {  
    int n;  
    if (!(in >> n)) { return false; }  
    for (int i = 0; i < n; i++) {  
        double val;  
        if (!(in >> val)) { return false; }  
        v.push_back(val);  
    }  
    return true;  
}
```

The caller, the caller's caller, etc. now need to be concerned whether this function returned true or false.

## Error handling with exceptions

```
void read_input(std::istream &in, std::vector<double> &v) {
    int n;
    if (!(in >> n))
        throw std::runtime_error("failed to read num elts");
    for (int i = 0; i < n; i++) {
        double val;
        if (!(in >> val))
            throw std::runtime_error("failed to read value");
        v.push_back(val);
    }
}
```



The called can just assume that either the function will either succeed completely, or will throw an exception.

*It is no longer the caller's responsibility to handle the possibility of failure. (Unless the caller wants to handle a failure.)*



## Using the read\_input function

```
// without exceptions
std::vector<double> data_vec;
if (!read_data(in, data_vec)) {
    // What are we supposed to do if the data can't be read
    // successfully? This might not be a good place to
    // report an error to the user.
}

// with exceptions
std::vector<double> data_vec;
read_data(in, data_vec);
// If we get here, we know the data was read successfully!
// If an exception was thrown, it is our CALLER's problem.
```

3. What keyword is used to generate an exception? What keyword indicates that the block of code may generate an exception? What keyword indicates what should be done in the case of an exception?

```
// generate an exception  
throw exception_object;  
  
// handle an exception  
try {  
    // ... this is code that might throw an exception ...  
} catch (exception_type &ex) {  
    // ... handle the possibility that an exception_type  
    //     exception was thrown ...  
    //  
}
```

4. In the case of multiple matching `catch` blocks for a thrown exception, which one actually catches the exception?

specific  $\longrightarrow$  general

The `catch` clauses of a `try` are checked in order. The first one that matches the type of the thrown exception is the one that is executed.

So, you should order your `catch` clauses in the order from most-specific (derived exception classes) to most-general (base exception classes.)

If your program defines custom exception types, it's a good idea to have them inherit from one of the "standard" exception classes (e.g., `std::runtime_error`.)

## 5. How do you get the message associated with an exception?

The standard exception classes (derived from `std::exception`) have a virtual member function called `what()` which returns a `std::string`.

This string is a text message describing the reason for the exception.

The constructors for the standard exception classes except a string value to set this message. E.g.

```
throw std::runtime_error("Couldn't open input file");
```

## Exercise 35

- Practice throwing and catching exceptions
- Breakout rooms 1–10 are social
- Use Slack to let us know if you have any questions!

## Day 36 recap questions

- 1 Why use iterators?
- 2 What are the bare minimum operators that need to be overloaded by an iterator?
- 3 When won't a simple pointer correctly iterate through a collection?
- 4 Given a container class, how/where should its iterator class be specified?
- 5 In addition to defining the iterator class, what else should the container do to support iterators?
- 6 What might go wrong if we don't also define a `const_iterator` for a container?

# 1. Why use iterators?

When implementing a *collection class* (a class whose instances are intended to store a collection of data values), implementing iterators provides a uniform way to access the data values in the collection.

Advantages:

- User of the container accesses its data using the same techniques as other container types
  - ... regardless of the underlying data structure
- Container works with STL algorithms and other generic functions designed to work with containers

## 2. What are the bare minimum operators that need to be overloaded by an iterator?

At a minimum:

- \* (dereference)
- ++ (advance)
- == and != (compare iterators)



### 3. When won't a simple pointer correctly iterate through a collection?

A pointer type works as an iterator only if the underlying storage for the values in the collection is an array.

If the collection uses a different data structure (such as a linked list), the member where elements are stored isn't guaranteed to be contiguous.

4. Given a container class, how/where should its iterator class be specified?

The `iterator` and `const_iterator` types for the collection must be *members* of the collection.

## Example of defining iterator types

```
class MyCollection {
public:
    class iterator {
        // ... members of iterator ...
    };

    class const_iterator {
        // ... members of const_iterator ...
    };

    iterator begin() { return /* begin iterator */ }
    iterator end() { return /* end iterator */ }

    const_iterator begin() const { return /* begin iterator */ }
    const_iterator end() const { return /* end iterator */ }

    // ...
}
```

## What an iterator type could look like

```
// Assume that EltType is the  
// element type of the collection  
class iterator {  
public:  
    // ...constructor(s), destructor...  
  
    EltType& operator*();  
    iterator& operator++();  
    bool operator!=(const iterator &rhs) const;  
    bool operator==(const iterator &rhs) const;  
  
private:  
    // ...member variables ...  
};
```

## 5. In addition to defining the iterator class, what else should the container do to support iterators?

The container should have appropriate `begin()` and `end()` member functions (as shown on previous slide.)

The `begin()` function returns an iterator (or `const_iterator`) positioned at the first element of the collection.

The `end()` function returns an iterator (or `const_iterator`) positioned just past the last element of the collection. (I.e., advancing an iterator which is positioned at the last element yields an iterator which is equal to the end iterator.)

## 6. What might go wrong if we don't also define a `const_iterator` for a container?

Since normal iterators can modify data values in the collection, they can't be used on a `const` collection object.

This is a problem if the collection is passed by `const` reference:

```
void analyze_data(const MyCollection &coll) {  
    // this won't work because coll is const  
    for (MyCollection::iterator i = coll.begin();  
         i != coll.end();  
         ++i) {  
        // ...  
    }  
}
```

## Exercise 36

- Implement an `iterator` type for the `MyList` class
- Breakout rooms 1–10 are “social”
- Use Slack to let us know if you have questions!

# Notes



# Notes

# Notes

# Notes

# Notes