Announcements

- No reading due this or next week
- Recitations:
  - This week: doxygen
  - Next week: Q&A (optional)
- PA8 will be assigned next Wednesday
  - But it's the efficiency assignment
  - You may use only standard C++
    - no additional libraries
    - You should not need -L
- Next week:
  - Monday: review + hand out code
  - Wednesday: midterm #2
  - Friday: midterm post-mortem

GTA surveys

- Every year, we ask students to evaluate their GTAs
  - You will evaluate me, too, via ASCSU
  - But later
- Go to the following surveymonkey site
  - https://www.surveymonkey.com/r/YC28DZ5
  - Select the GTA you have recitations with
  - Fill out the survey (please)

A few more notes on efficiency

- Compiler optimization flags
  - Let the compiler help you!
  - -O2 is fast, safe.
  - -O3 is potentially risky
  - Works better, the more you use const
- Minimize memory consumption
- Keep your unit tests and reference implementation
  - Compare results
  - It’s easy to insert bugs while optimizing!

Return Value Optimization

- Least Efficient
  Quagga q;
  q.name = "fred";
  return q;
- More efficient
  Quagga q("Fred");
  return q;
- More efficient still
  return (Quagga q("fred"));
- This allows the compiler to make q in the memory of the receiving function...

OO in C++ vs Java

- C++ separates declaration from implementation
  - To support separate compilation
- C++ has destructors
  - Because users manage their own memory
- C++ has explicit accessors & mutators
  - Because of const
- C++ allows operator overloading
- C++ supports both dynamic & static dispatch
  - Static is the dispatch
- C++ allows multiple inheritance
- C++ allows no parent class
  - Java has a single, rooted inheritance tree
- C++ interfaces are declared using pure virtual methods
**OO Issues Special to C++**

- Memory Management
  - Big '3': deep versions of
    - Copy constructor
    - Destructor
    - Assignment operator
  - When classes have pointer to the heap
- Slicing
  - Containers of objects, not just references
  - Do not mix well with polymorphism
- Beware of
  - Virtual methods in constructors
  - Non-virtual destructors

**Odds & Ends**

- Some minor aspects of OO programming in C++ that we haven’t talked about...
  - Friends
  - Static methods
  - Static data fields
  - Incomplete class definitions
- Other topics?

**Friend (example)**

```cpp
Class ListNode {
private:
    int element;
    ListNode* next;
    // Note private constructor
    ListNode(int theElement, ListNode* n = NULL):
        element(theElement), next(n) {}

    friend class IntQueue;
};
```

**Friend**

- Friends are exceptions to access schemes
- If class A declares class B to be a friend
  - B can access the private data in A
  - Note that 'friend' gives access
    - It doesn’t take it
- In the previous example
  - ListNode’s constructor is private
  - So only IntQueues and ListNodes can make ListNodes
- Friendship is not inherited
  - Private data of classes that inherit A is not visible to B

**Methods as Friends**

```cpp
Class ListNode {
private:
    int element;
    ListNode* next;
    // Note private constructor
    ListNode(int theElement, ListNode* n = NULL):
        element(theElement), next(n) {}

    friend void IntQueue::Push(int v);
};
```

**Unsolicited Opinion**

- 'Friend' is usually poor style in C++
  - Used as a fix for a poor OO design
  - Many style manuals ban it
- Avoid it whenever possible
  - OK as a patch before refactoring
  - OK with private constructor design pattern
- If you must use it:
  - Comment why (extensively)
  - Friend methods do less damage than Friend classes
Static Methods (Example)

```cpp
class Math {
public:
    static double sqrt(double input);
};
```

Static Methods

- Static declaration precedes return type
  - Like 'virtual' or 'inline'
- Static methods have no hidden argument
  - Therefore, they cannot access object fields
  - They cannot be declared const
    - i.e. no const after parameter list
  - Because there is no hidden variable to be const
- Static methods are stand-alone functions
  - An alternative is to just make them functions
  - Therefore, less common in C++ than Java
  - Semantically, static methods are bound to their class
    - As in the Math::sqrt example
    - In essence, class is used like a namespace

Static Data Fields (Example)

```cpp
class CountedObject {
public:
    CountedObject() { ctr++; }
    virtual ~CountedObject() { ctr--; }
    static int Count() const { return ctr; }
protected:
    static int ctr;
};
```

-----------------------------
```
// CountedObject.cpp
// initialize static data field
CountedObject::ctr = 0;
```

Static Data Fields

- Most data fields are stored inside objects
- Static data fields are stored once per class
  - They are access-controlled global variables
  - Never use global variables. Static fields are better
  - Initialized in .cpp files
    - Initialization will happen before main is entered
- Useful for "per class" data
  - E.g. reference counting
    - Previous example
  - Efficient way of storing class properties
    - Every Quagga has the same body temperature, so...

Incomplete Class Definition
(Example #1)

```cpp
class OddListElement;
```

```cpp
class EvenListElement {
public:
    void Push();
protected:
    OddListElement* next;
};
```

```cpp
class OddListElement {
public:
    void Push();
protected:
    EvenListElement* next;
};
```

Incomplete Class Declarations

- Allow a class to be used before it is declared
  - Useful for recursive data structures
- Limited to situations where the size of the class doesn't matter
  - In previous example, we declared a pointer
  - We could also have declared a reference
  - The body of Push() can use real objects
    - Because both have been declared before Push() is implemented (in .cpp file)
Incomplete Class Declaration
(Example #2)

class Quagga;
class Herd {
public:
    void Join(Quagga* new_member);
protected:
    vector<Quagga*> members;
};

// Herd.cpp
#include<Herd.h>
#include<Quagga.h>
// Now implement Join...

More Incomplete Declarations

- If nothing in class A's .h file needs to know the size of B
  - But it uses pointers or references to B's
- Then put an incomplete declaration of B in A.h
- And include B.h in A.cpp
- This reduces compilation dependencies
  - A.h no longer depends on B.h
  - Therefore other files that include A.h don't on B.h
    - Although A.cpp does
- If B.h is changed, fewer files need to be recompiled
  - Requires correct dependencies in Makefile