CSE 1325: Object-Oriented Programming

Lecture 7 – Chapter 10

Multiple Inheritance, Input / Output, and Files

Mr. George F. Rice

george.rice@uta.edu

Based on material by Bjarne Stroustrup

www.stroustrup.com/Programming

ERB 402

Office Hours:

Tuesday Thursday 11 - 12

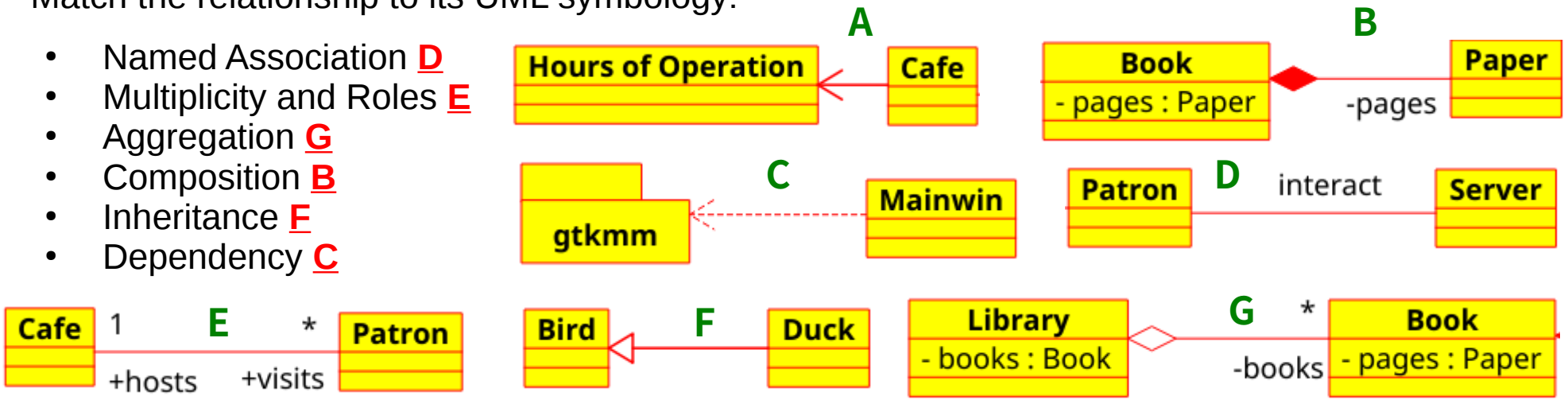
Or by appointment

Lecture 6
Quick Review

- The PIE definition of Object-Oriented Programming posits 3 foundations: **Encapsulation (data hiding)**, **Inheritance (reuse and extension of base class members)**, and **Polymorphism (TBD)**. Define 2 of the 3.
- **True** or False: A class is a user-defined type.
- **True** or False: The only difference in C++ between a struct and a class is the default visibility of its members.
- True or **False**: A class (type) may not be used to instantiate templates such as vector
- What is a friend function? **A non-member function granted access to private members.**
- True or **False**: A friend function is a member of the class with which it is friends.
- Ensuring that a program operates on clean, correct and useful data is **data validation**.
- Algorithmically enforceable constraints on the correctness, meaningfulness, and security of input data are **validation rules**.
- Why should classes be designed to guarantee that the data they encapsulate is valid? **Avoid constant revalidation and avoid bugs caused by invalid data**
- True or **False**: An enum class is just an enum that also permits methods.
- What is the best way to print out (stream) an enum class variable? **Operator overloading**

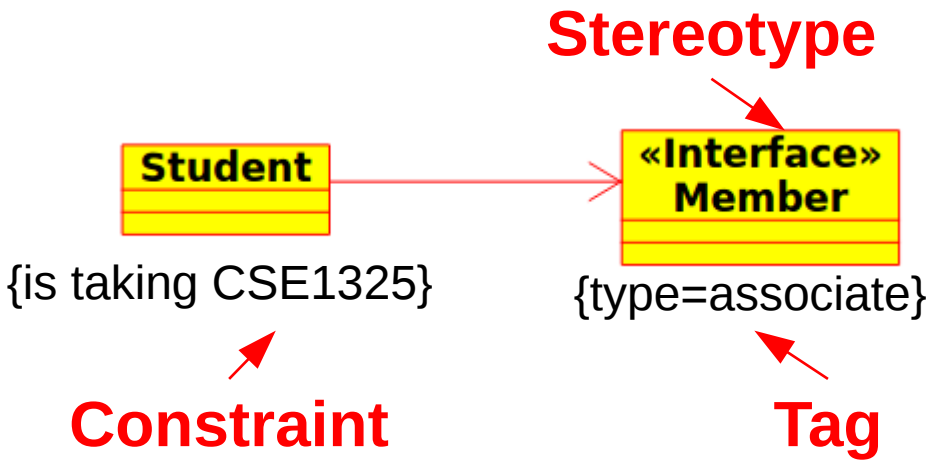
Lecture 6
Quick Review

- Providing a user-defined meaning to a pre-defined operator (e.g., +, ==, <<) for a user-defined type (class) is **operator overloading**.
- To overload the + operator for a class, define the **operator+** method.
- True or **False**: C++ allows you to define new operators, such as @@.
- Reuse and extension of fields and method implementations from one class to other classes is called **inheritance**. The original class is called the **base** class, and the other classes are called **derived** classes.
- A **class hierarchy** defines the inheritance relationships between classes.
- In specifying a base class in the derived class, the keyword “public”, “protected”, or “private” is specified. What are the implications of this? **Visibility limits of inherited members**
- Match the relationship to its UML symbology:
  - Named Association **D**
  - Multiplicity and Roles **E**
  - Aggregation **G**
  - Composition **B**
  - Inheritance **F**
  - Dependency **C**



Quick Review

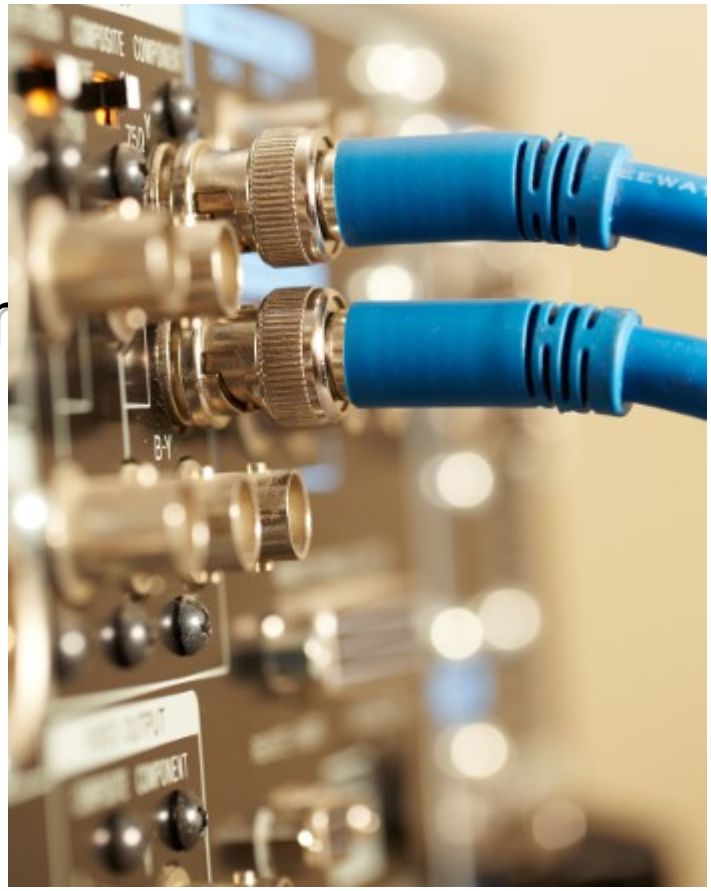
- Identify the 3 UML extensions is this diagram. What do they mean? Anything the author wants them to mean



- A series of interactions that enable an actor to achieve a goal is a **use case**.
- True or false: In UML, a use case is treated as a class. **True**

Overview

- Simple Inheritance Redux
- Multiple Inheritance
  - Practical Uses
  - Resolving the “diamond problem”
- Fundamental I/O concepts
  - Files and streams
  - Handling I/O errors
  - Managing stream status



A Quick Review of Simple Inheritance

```
ricegf@pluto:~/dev/cpp/201801/07$ make barnyard_simple
g++ --std=c++14 -c barnyard_simple.cpp
g++ --std=c++14 -o barnyard_simple barnyard_simple.o
ricegf@pluto:~/dev/cpp/201801/07$ ./barnyard_simple
W E L C O M E   T O   T H E   B A R N Y A R D !
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
```

```
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>
using namespace std;

class Critter {
  public:
    Critter(int frequency) : _frequency{frequency}, _timer{0} { }
    void count() {if (++_timer > _frequency) _timer = 0;}
    void speak() { if (!_timer) cout << "Generic critter sound!" << endl; }
  protected:
    int _frequency;
    int _timer;
};
int main() {
  vector<Critter> critters{Critter{13}, Critter{11}, Critter{7}, Critter{3}};

  cout << "W E L C O M E   T O   T H E   B A R N Y A R D !" << endl;
  for (int i=0; i<120; ++i) {
    for (Critter& c: critters) { c.count(); c.speak(); }
    this_thread::sleep_for(chrono::milliseconds(50));
  }
}
```

Timer is just a counter. When expired, the critter makes a sound.
Frequency is how many calls to count() between sounds.

The above idiom pauses the program for 50 milliseconds, or 0.05 seconds

A Quick Review of Simple Inheritance

```
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>
using namespace std;

class Critter {
  public:
    Critter(int frequency) : _frequency{frequency}, _timer{0} { }
    void count() {if (++_timer > _frequency) _timer = 0;}
    void speak() { if (!_timer) cout << "Generic critter sound!" << endl; }
  protected:
    int _frequency;
    int _timer;
};
int main() {
  vector<Critter> critters{Critter{13}, Critter{11}, Critter{7}, Critter{3}};

  cout << "W E L C O M E   T O   T H E   B A R N Y A R D !" << endl;
  for (int i=0; i<120; ++i) {
    for (Critter& c: critters) { c.count(); c.speak(); }
    this_thread::sleep_for(chrono::milliseconds(50));
  }
}
```

What code needs to change to add barnyard animals (cow, chicken, dog...)?

Using Inheritance

```
#include <iostream>, <vector>, <chrono>, <thread>
using namespace std;

class Critter {
  public:
    Critter(int frequency) : _frequency{frequency}, _timer{frequency} { }
    void count() {if (++_timer > _frequency) _timer = 0;}
    void speak() { if (!_timer) cout << "Generic critter sound!" << endl; }
  protected:
    int _frequency;
    int _timer;
};

class Cow : public Critter {
  public:
    Cow(int frequency) : Critter(frequency) { }
    void speak() { if (!_timer) cout << "Moo! Mooooo!" << endl; }
};
class Dog : public Critter {
  public:
    Dog(int frequency) : Critter(frequency) { }
    void speak() { if (!_timer) cout << "Woof! Woof!" << endl; }
};
class Chicken : public Critter {
  public:
    Chicken(int frequency) : Critter(frequency) { }
    void speak() { if (!_timer) cout << "Cluck! Cluck!" << endl; }
};
```

The base class remains unchanged.
The derived classes inherit the implementation of count.

Reusing Methods with Inheritance

```
int main() {
  vector<Dog> dogs = {Dog{11}, Dog{9}, Dog{3}};
  vector<Cow> cows = {Cow{7}, Cow{13}};
  vector<Chicken> chickens{Chicken{2}, Chicken{5}};

  cout << "W E L C O M E   T O   T H E   B A R N Y A R D !" << endl;
  for (int i=0; i<120; ++i) {
    for (auto& c: dogs) { c.count(); c.speak(); }
    for (auto& c: cows) { c.count(); c.speak(); }
    for (auto& c: chickens) { c.count(); c.speak(); }
    this_thread::sleep_for(chrono::milliseconds(50));
  }
}
```

The Dog, Cow, and Chicken classes all used the count() method from Critter.

```
ricegf@pluto:~/dev/cpp/201801/07$ make barnyard_animals
g++ --std=c++14 -c barnyard_animals.cpp
g++ --std=c++14 -o barnyard_animals barnyard_animals.o
ricegf@pluto:~/dev/cpp/201801/07$ ./barnyard_animals
W E L C O M E   T O   T H E   B A R N Y A R D !
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
Cluck! Cluck!
Woof! Woof!
Moo! Mooooo!
Cluck! Cluck!
Woof! Woof!
```

Simplify?

- It's awkward to keep a separate vector for each derived type
- Why not keep a single vector of type Critter – since, after all, Dog, Cow, and Chicken "isa" Critter! Right?
- Well, sure you can – but it's a lot harder in C++ than in most other languages (*sigh*)
  - Your vector must contain *pointers* to Critters and its derivations, not actual objects
    - Or references – but that's actually *more* awkward
  - Critter::speak must be declared *virtual*
    - The base class must give *explicit permission* to override its methods

Preview of Coming Attractions

Reusing Methods with Inheritance

```
#include <iostream>, <vector>, <chrono>, <thread>
using namespace std;

class Critter {
  public:
    Critter(int frequency) : _frequency{frequency}, _timer{frequency} { }
    void count() {if (++_timer > _frequency) _timer = 0;}
    virtual void speak() { if (!_timer) cout << "Generic critter sound!" << endl; }
  protected:
    int _frequency;
    int _timer;
};

class Cow : public Critter {
  public:
    Cow(int frequency) : Critter(frequency) { }
    void speak() { if (!_timer) cout << "Moo! Mooooo!" << endl; }
};
class Dog : public Critter {
  public:
    Dog(int frequency) : Critter(frequency) { }
    void speak() { if (!_timer) cout << "Woof! Woof!" << endl; }
};
class Chicken : public Critter {
  public:
    Chicken(int frequency) : Critter(frequency) { }
    void speak() { if (!_timer) cout << "Cluck! Cluck!" << endl; }
};
```

Virtual allows the method of a *derived* type to be accessed via a variable of *this* type, i.e., *polymorphically*

Preview of Coming Attractions

Reusing Methods with Inheritance

```
int main() {
  vector<Critter*> critters = {new Dog{11}, new Dog{9}, new Dog{3},
                               new Cow{7},  new Cow{13},
                               new Chicken{2}, new Chicken{5}};

  cout << "W E L C O M E   T O   T H E   B A R N Y A R D !" << endl;
  for (int i=0; i<120; ++i) {
    for (auto c: critters) { c->count(); c->speak(); }
    this_thread::sleep_for(chrono::milliseconds(50));
  }
}
```

Pointer to Critter

"new" instances Dog et. al. and returns a pointer to the instance

-> accesses a class member via a pointer

So yes, we can simplify – but, it's complicated.*

We'll cover polymorphism formally in lecture 19. This is a preview of coming attractions.

Polymorphism won't be on the first exam.

```
ricegf@pluto:~/dev/cpp/201801/07$ make barnyard_animals_poly
g++ --std=c++14 -c barnyard_animals_poly.cpp
g++ --std=c++14 -o barnyard_animals_poly barnyard_animals_poly.o
ricegf@pluto:~/dev/cpp/201801/07$ ./barnyard_animals_poly
W E L C O M E   T O   T H E   B A R N Y A R D !
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
Cluck! Cluck!
Woof! Woof!
Moo! Mooooo!
Cluck! Cluck!
```

* Yes, the irony is not lost on me. Trust me.

Multiple Inheritance

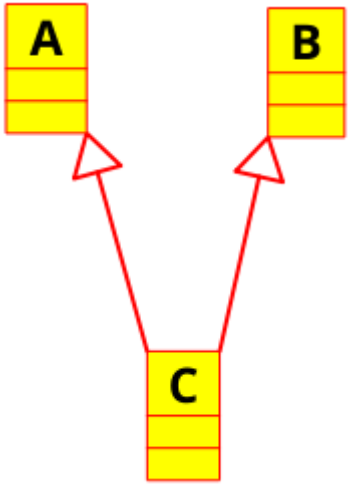
- What if a class is derived from more than one base class?
  - This is called *multiple inheritance*
  - You inherited traits from both your biological mother and father* – *multiple inheritance*
- With multiple inheritance, each base class's members are laid out sequentially in memory after the derived class's members

Multiple Inheritance is a derived class inheriting class members from two or more base classes.

* Unless you're a clone, in which case – *single inheritance*!

Multiple Inheritance in UML and C++

- In UML, just use multiple arrows



- In C++, just list multiple base classes

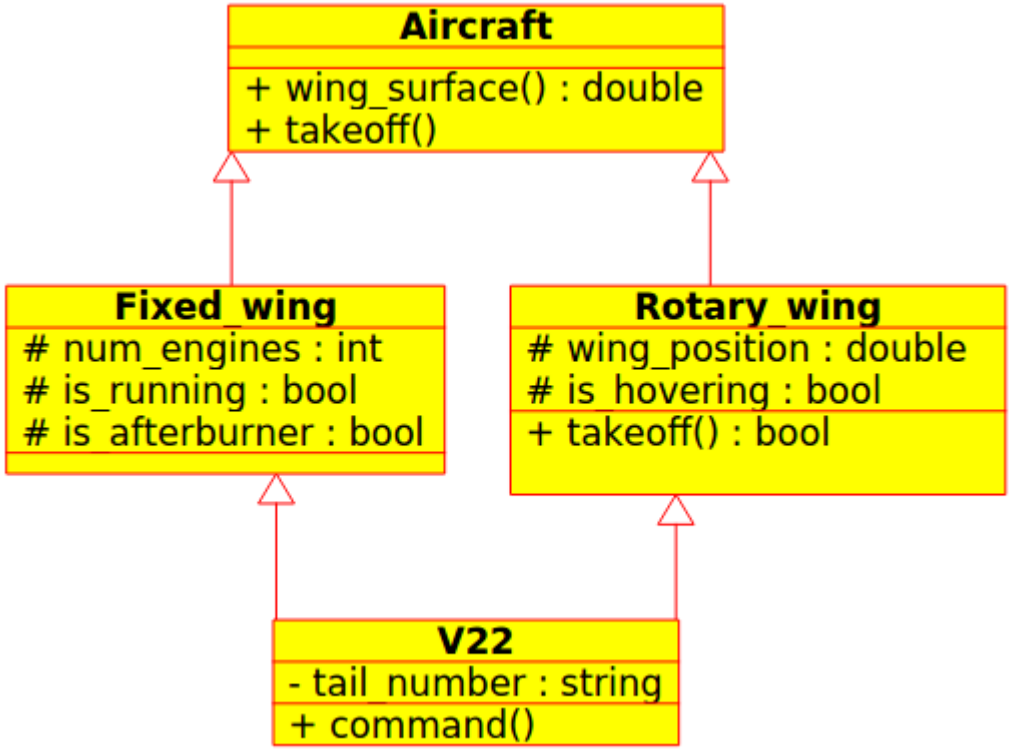
```
#include<iostream>
using namespace std;

class A {
  public: A()  { cout << "A's constructor called" << endl; }
};

class B {
  public: B()  { cout << "B's constructor called" << endl; }
};

class C: public A, public B {
  public: C()  { cout << "C's constructor called" << endl; }
};

int main() {
    C c;
}
```

```
ricegf@pluto:~/dev/cpp/201801/07$ make mi
g++ --std=c++14 -c mi.cpp
g++ --std=c++14 -o mi mi.o
ricegf@pluto:~/dev/cpp/201801/07$ ./mi
A's constructor called
B's constructor called
C's constructor called
ricegf@pluto:~/dev/cpp/201801/07$
```

Constructors are called in the order listed.
Destructors are called in the reverse order listed.

Multiple Inheritance

Example with Multiple Inheritance



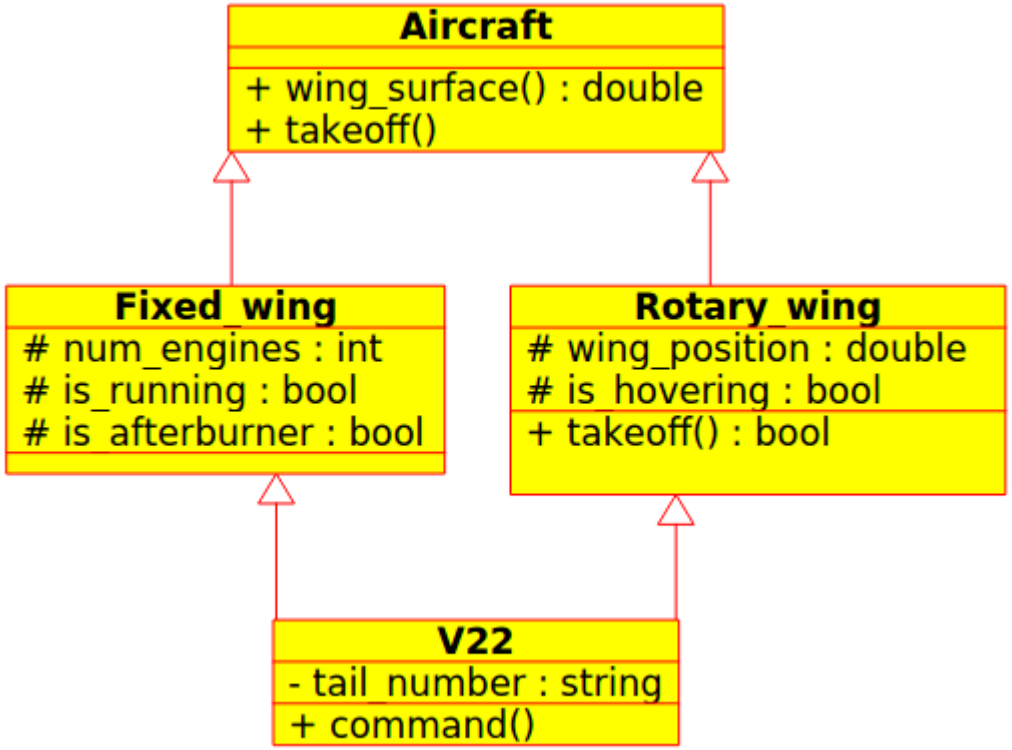
V22 would have wing_surface(), takeoff(), and command() methods

Wait a minute…
***which* takeoff()?**

And, more subtly, both the Fixed_wing and Rotary_wing classes inherit wing_surface(). Which path will inheritance follow, i.e., *which* inherited wing_surface() will V22 invoke?

https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem
http://www.cprogramming.com/tutorial/multiple_inheritance.html
http://www.cprogramming.com/tutorial/virtual_inheritance.html

Example with Multiple Inheritance



This is called the "**diamond problem**" - when inheriting in a "diamond shape", inheritance paths are no longer obvious.

Java, C#, and Ruby (for example) sidestep this problem and do **NOT** implement multiple inheritance*, using Interfaces (also called Protocols) instead.

Python makes inheritance order significant, and calls the first method found – left to right, then bottom to top.

How does C++ react?

https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem
http://www.cprogramming.com/tutorial/multiple_inheritance.html
http://www.cprogramming.com/tutorial/virtual_inheritance.html

* Java SE 8 allows inheriting interface definitions, which is *sorta* multiple inheritance.

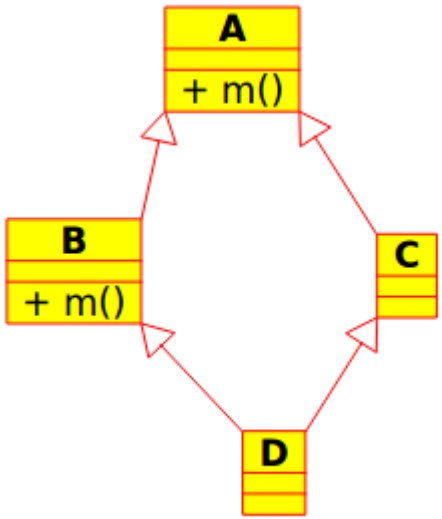
The ABCD Diamond

```
#include <iostream>
using namespace std;

class A {
  public:
    virtual void m() {cout << "m of A" << endl;}
};

class B : public A {
  public:
    virtual void m() override {cout << "m of B" << endl;}
};

class C : public A { };

class D : public B, public C { };

int main() {
  D d;
  d.m();
}
```

C++ (for once) refuses to make an assumption and instead raises a compiler error.

```
ricegf@pluto:~/dev/cpp/201701/17$ g++ -std=c++11 diamond.cpp
diamond.cpp: In function ‘int main()’:
diamond.cpp:22:5: error: request for member ‘m’ is ambiguous
   d.m();
     ^
diamond.cpp:6:18: note: candidates are: virtual void A::m()
     virtual void m() {cout << "m of A" << endl;}
                  ^
diamond.cpp:11:18: note:                 virtual void B::m()
     virtual void m() override {cout << "m of B" << endl;}
                  ^
ricegf@pluto:~/dev/cpp/201701/17$
```

Let’s simplify and test (a GREAT strategy for learning a new language!).



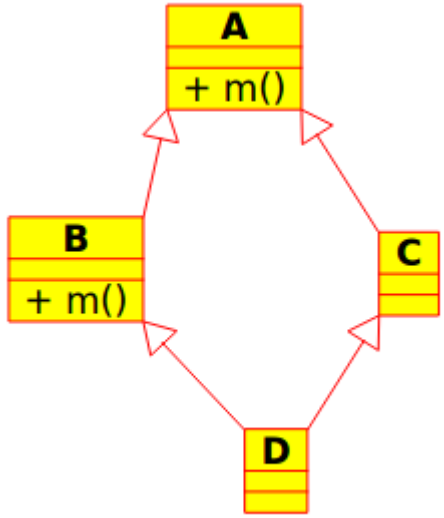
Explicit Base Class Method Call

```
#include <iostream>
using namespace std;

class A {
  public:
    virtual void m() {cout << "m of A" << endl;}
};

class B : public A {
  public:
    virtual void m() override {cout << "m of B" << endl;}
};

class C : public A { };

class D : public B, public C { };

int main() {
  D d;
  d.B::m();
  d.C::m();
}
```

Ambiguity resolved!

```
ricegf@pluto:~/dev/cpp/201701/17$ g++ -std=c++11 diamond.cpp
ricegf@pluto:~/dev/cpp/201701/17$ ./a.out
m of B
m of A
ricegf@pluto:~/dev/cpp/201701/17$
```

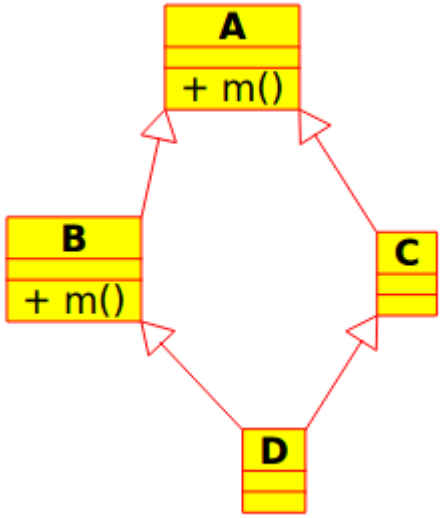
What if we specify the method using namespace notation?



The ABCD Diamond

```
#include <iostream>
using namespace std;

class A {
  public:
    virtual void m() {cout << "m of A" << endl;}
};

class B : public A {
  public:
    virtual void m() override {cout << "m of B" << endl;}
};

class C : public A { };

class D : public B, public C { };

int main() {
  D d;
  d.A::m();
  d.B::m();
  d.C::m();
}
```

Let's explicitly call A's method, then...



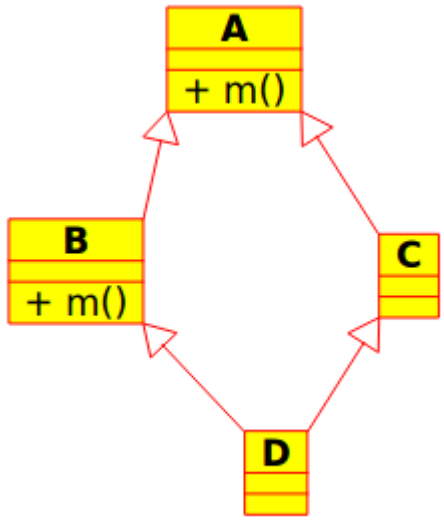
Uh oh

```
ricegf@pluto:~/dev/cpp/201701/17$ g++ -std=c++11 diamond.cpp
diamond.cpp: In function 'int main()':
diamond.cpp:20:8: error: 'A' is an ambiguous base of 'D'
   d.A::m();
        ^
ricegf@pluto:~/dev/cpp/201701/17$
```

Virtual Inheritance

```
#include <iostream>
using namespace std;

class A {
  public:
    virtual void m() {cout << "m of A" << endl;}
};

class B : public virtual A {
  public:
    virtual void m() override {cout << "m of B" << endl;}
};

class C : public virtual A { };

class D : public B, public C { };

int main() {
  D d;
  d.A::m();
  d.B::m();
  d.C::m();
}
```

Easier done than said.
Just make the intermediate classes inherit *virtually*.




```
ricegf@pluto:~/dev/cpp/201701/17$ g++ -std=c++11 diamond.cpp
ricegf@pluto:~/dev/cpp/201701/17$ ./a.out
m of A
m of B
m of A
ricegf@pluto:~/dev/cpp/201701/17$
```

We'll talk more about the virtual keyword, and what it *really* does, in Lecture 17.

Pure virtual functions

- Often, a function in an interface can't be implemented
  - E.g. the data needed is "hidden" in the derived class
  - We must ensure that a derived class implements that function
  - Make it a "pure virtual function" (**=0**)
- This is how we define truly abstract interfaces ("pure interfaces")

```
// interface ONLY
class Engine {
    // no data
    // (usually) no constructor
    virtual double increase(int i) = 0; // must be defined in a derived class
    // …
    // (usually) a virtual destructor
    virtual ~Engine();
};

Engine eee;         // error: Collection is an abstract class
```

Pure virtual functions

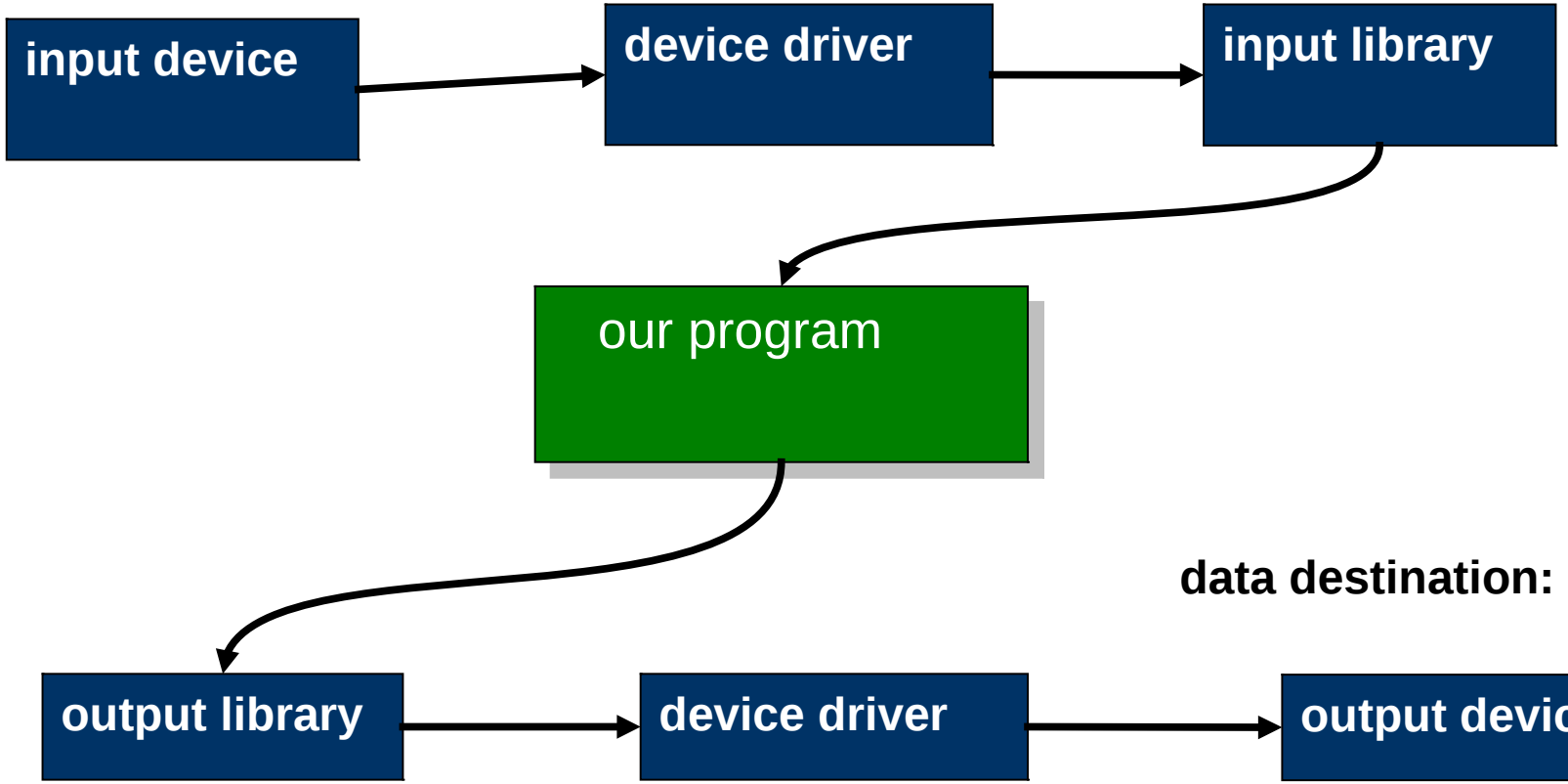
- A pure interface can then be used as a base class

```
class M123 : public Engine {
  public:
    M123();  // construtor:  initialization, acquire resources
    double increase(int i) { /* … */ }   // overrides Engine::increase
    // …
    ~M123(); // destructor
};

M123 window3_control; // OK
```

Switching Gears Now
To I/O and Files

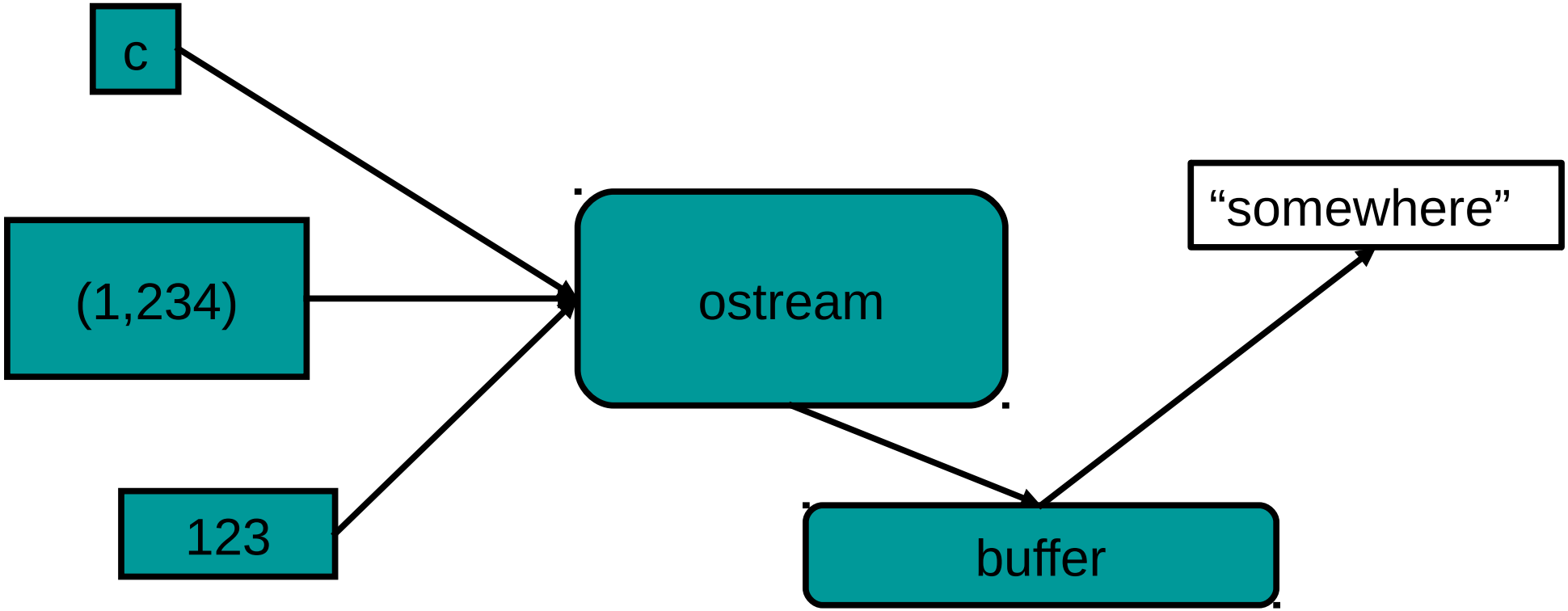
Input and Output

data source:



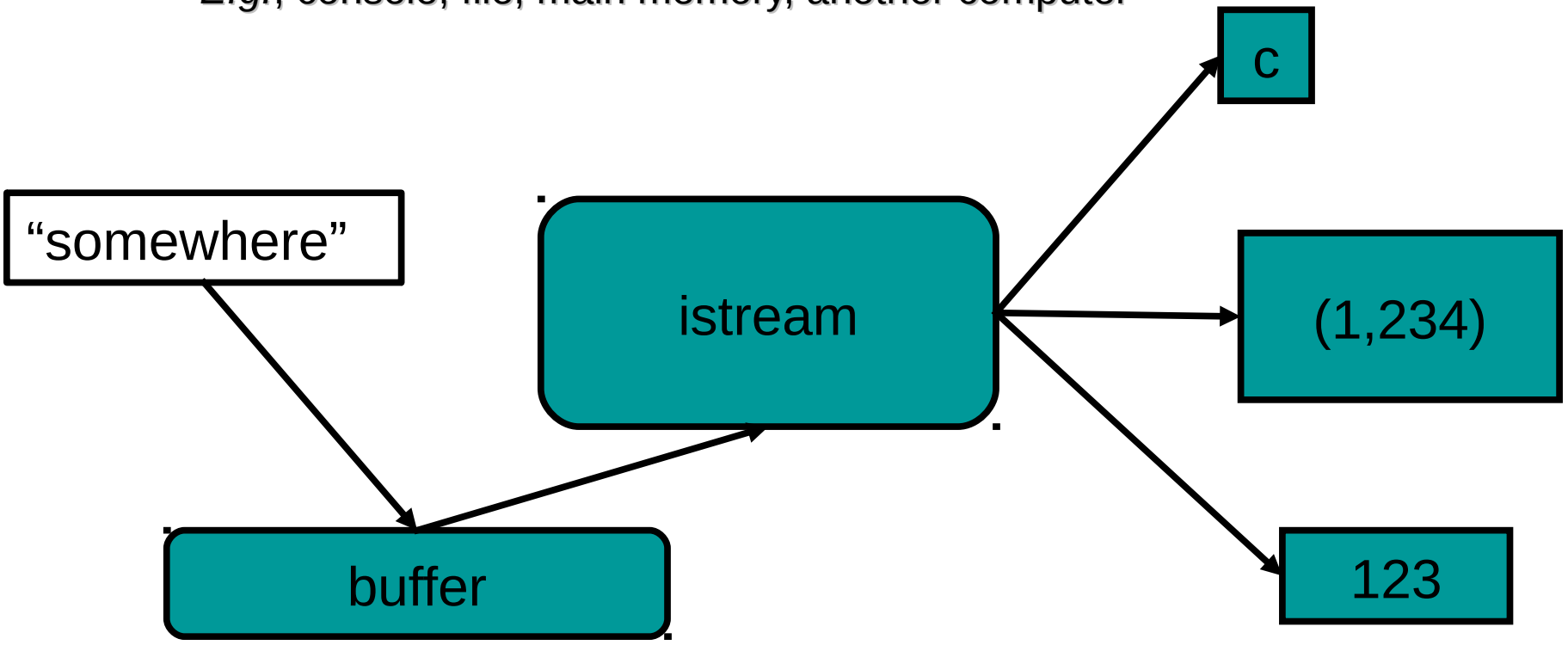
The stream model

- An **ostream**
  - turns values of various types into character sequences
  - sends those characters somewhere
    - *E.g.*, console, file, main memory, another computer



The stream model

- An **istream**
  - turns character sequences into values of various types
  - gets those characters from somewhere
    - *E.g.*, console, file, main memory, another computer



Whither the Stream?

- What you do with the data in the stream determines the program flow
- Example: Handling an XML file
  - DOM: The XML stream's tags are interpreted to build a complete Document Object Model structure containing all the data in memory
  - SAX: The XML stream's tags trigger events – calls to your program's methods to handle each tag

```
<note>
<to>A. Student</to>
<from>Prof Rice</from>
<heading>Reminder</heading>
<body>Exam #3 is Apr 13.</body>
</note>
```

- Both approaches are valid and will parse the XML file, but the code looks *very* different

The stream model

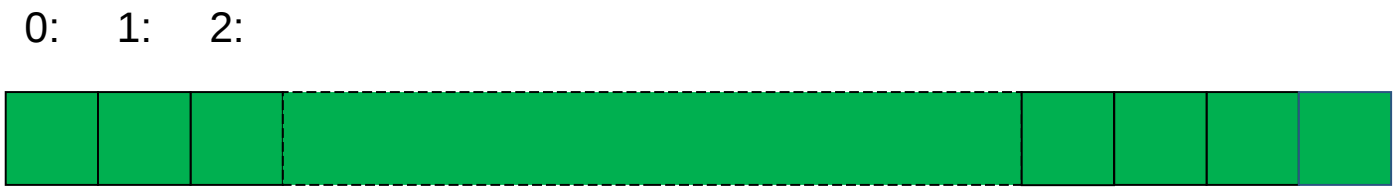
- Reading and writing
  - Of typed entities
    - << (output) and >> (input) plus other operations
    - Type safe
    - Formatted
  - Typically stored (entered, printed, etc.) as text
    - But not necessarily (see binary streams in chapter 11)
  - Extensible
    - You can define your own I/O operations for your own types
  - A stream can be attached to any I/O or storage device

Files

- We turn our computers on and off
  - The contents of our main memory is transient
- We like to keep our data
  - So we keep what we want to preserve on disks, solid-state drives, cloud drives, and similar permanent storage
- A file is a self-contained named sequence of bytes available via the operating system
  - A file has a name
  - The file has a data format
- We can read/write a file if we know its name and format
  - Not always a given, e.g., early Microsoft Office file formats

File – A self-contained named sequence of bytes available via the operating system

A File



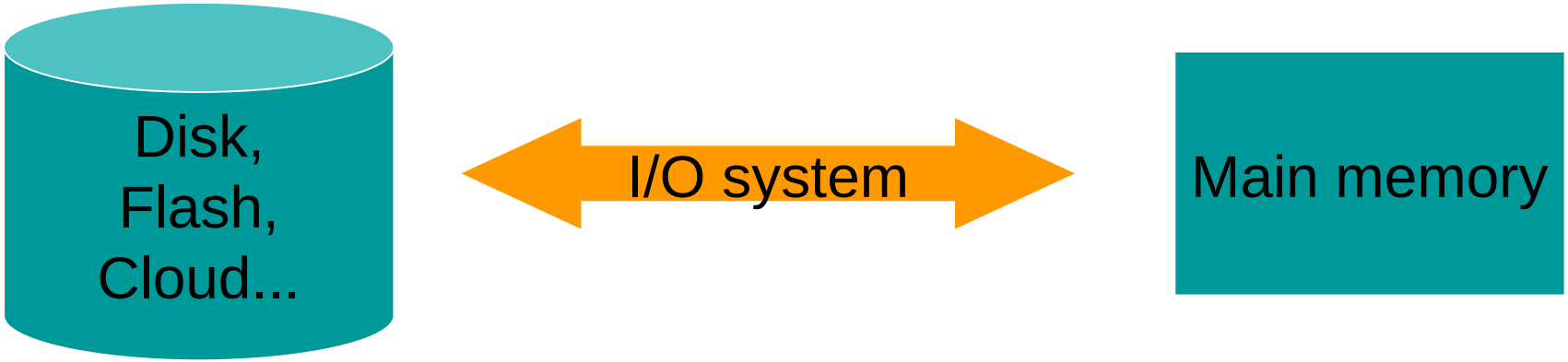
- At the fundamental level, a file is a sequence of bytes numbered from 0 upwards
- Other notions can be supplied by programs that interpret a “file format”
  - For example, the 6 bytes "123.45" might be interpreted as the floating-point number 123.45

You can examine binary files

In Linux and Unix, “hexdump -C [file]” will reveal the contents of a binary file
Windows requires a hex editor app or a Powershell script – GIYF

Files

- General model



Files
(sequences of bytes)

iostreams

Objects
(of various types)

Files

- To read a file
  - We must know its name
  - We must open it (for reading)
  - Then we can read
  - Then we must close it
    - That is typically done implicitly
- To write a file
  - We must know its name
    - Or decide on a name for a new file
  - We must open it (for writing)
    - Or create a new file of that name
  - Then we can write it
  - We must close it
    - That is typically done implicitly

Opening a file for reading

```
ricegf@pluto:~/dev/cpp/201801/07$ make read
g++ --std=c++14 -c read.cpp
g++ --std=c++14 -o read read.o
ricegf@pluto:~/dev/cpp/201801/07$ ./read
Please enter input file name: read.cpp
#include <iostream>
#include <fstream>
#include <stdexcept>

using namespace std;
```

```
// …
int main()
{
    cout << "Please enter input file name: ";
    string iname;
    cin >> iname;
    // The following line only works in later C++ versions, e.g., 11
    ifstream ist {iname}; // ifstream is an“input stream from a file”
                 // defining an ifstream with a name string
                 // opens the file of that name for reading
    if (!ist) throw runtime_error("can’t open input file " + iname);
    string aline;
    while (getline(ist, aline)) cout << aline << endl;
}
```

http://www.cplusplus.com/reference/fstream/ifstream/

Opening a file for writing

```
ricegf@pluto:~/dev/cpp/201801/07$ make write
g++ --std=c++14 -c write.cpp
g++ --std=c++14 -o write write.o
ricegf@pluto:~/dev/cpp/201801/07$ ./write
Please enter name of output file: temp.txt
ricegf@pluto:~/dev/cpp/201801/07$ cat temp.txt
Writing this to a file.
ricegf@pluto:~/dev/cpp/201801/07$
```

```
#include <iostream>
#include <fstream>
#include <stdexcept>
using namespace std;

int main () {
  cout << "Please enter name of output file: ";
  string oname;
  cin >> oname;

  ofstream ofs {oname};
  if (!ofs) throw runtime_error("can't open output file " + oname);

  ofs << "Writing this to a file.\n";
  return 0;
}
```

Reading from a file

- Suppose a file contains a sequence of pairs representing hours and temperature readings

  0 60.7
  1 60.6
  2 60.3
  3 59.22
- The hours are numbered **0..23**
- No further format is assumed
  - Maybe we can do better than that (but not just now)
- Termination
  - Reaching the end of file terminates the read
  - Anything unexpected in the file terminates the read
    - *E.g.*, **q**

Reading a File

```
#include <iostream>
#include <vector>
#include <stdexcept>
using namespace std;

class Reading { // a temperature reading
    public:
        Reading(int p_hour, double p_temp)
            : hour(p_hour), temperature(p_temp) {}
        int get_hour() {return hour;}
        double get_temp() {return temperature;}
    private:
        int hour;       // hour after midnight [0:23]
        double temperature;
};

int main() {
        vector<Reading> temps;  // create a vector to store the readings

        int hour;
        double temperature;
        while (cin >> hour >> temperature) {                    // read
                if (hour < 0 || 23 <hour)
                    throw runtime_error("hour out of range");   // check
                temps.push_back(Reading(hour,temperature));     // store
        }
}
```

Reading a File Redux

Most logic is moved into the class (including data validation!)

```
#include <iostream>
#include <vector>
#include <stdexcept>
using namespace std;

class Reading { // a temperature reading
    public:
        Reading(int p_hour, double p_temp)
            : hour(p_hour), temperature(p_temp) {validate();}
        int get_hour() {return hour;}
        double get_temp() {return temperature;}
        friend istream &operator>>(istream  &ist, Reading &r);
        void validate() {
          if (hour < 0 || 23 < hour) throw runtime_error("hour out of range");
        }
    private:
        int hour;       // hour after midnight [0:23]
        double temperature;
};
istream& operator>>(istream  &ist, Reading &r) {
  ist >> r.hour >> r.temperature;
  r.validate();
  return ist;
}
int main(){
        vector<Reading> temps;       // create a vector to store the readings
        Reading reading{0,0};        // temporary to hold input
        while (cin >> reading) temps.push_back(reading);  // read and store
}
```

I/O Error Handling

- Sources of errors
  - Human mistakes
  - Files that fail to meet specifications
  - Specifications that fail to match reality
  - Programmer errors
  - Etc.

- iostream reduces all errors to one of four states
  - **good()** *// the operation succeeded*
  - **eof()** *// we hit the end of input ("end of file")*
  - **fail()** *// something unexpected happened (including format error)*
  - **bad()** *// something unexpected and serious happened*

- Note that good() and bad() are **not** opposites, despite the name
  - Name selection for these methods was especially tortured...

Sample integer read “failure”

- Ended by “terminator character”
  - 1 2 3 4 5 *
  - State is **fail()**
- Ended by format error
  - 1 2 3 4 5.6
  - State is **fail()**
- Ended by “end of file”
  - 1 2 3 4 5 end of file
  - 1 2 3 4 5 Control-Z (Windows)
  - 1 2 3 4 5 Control-D (Mac* / Linux)
  - State is **eof()**
- Something really bad
  - Disk format error
  - State is **bad()**

* A second Control-D at the start of a line may also be required

I/O error handling

```
void fill_vector(istream& ist, vector<int>& v, char terminator)
{       // read integers from ist into v until we reach eof() or terminator
    for (int i; ist >> i; )   // read until “some failure”
        v.push_back(i);  // store in v
    if (ist.eof()) return;              // fine: we found the end of file
    if (ist.bad())
       throw runtime_error("ist is bad"); // stream corrupted; let’s get out of here!

    if (ist.fail()) {       // clean up the mess as best we can and report the problem
        ist.clear();         // clear stream state, so that we can look for terminator
        char c;
        ist >> c;              // read a character, hopefully terminator
        if (c != terminator) {            // unexpected character
            ist.unget();          // put that character back
            ist.clear(ios_base::failbit); // set the state back to fail()
        }
    }
}
```

Throwing I/O Exception on Bad()

- Request that a stream throw an exception if it goes bad

```
// How to make ist throw an exception if the stream goes bad:
ist.exceptions(ist.exceptions() | ios_base::badbit); // add badbit to exception mask
```

- This allows us to simplify input to this

```
void fill_vector(istream& ist, vector<int>& v, char terminator)
{       // read integers from ist into v until we reach eof() or terminator
    for (int i; ist >> i; )
        v.push_back(i);
    if (ist.eof()) return;    // fine: we found the end of file

    // not good() and not bad() and not eof(), ist must be fail()
    ist.clear();     // clear stream state
    char c;
    ist >> c;        // read a character, hopefully terminator
    if (c != terminator) {    // ouch: not the terminator, so we must fail
        ist.unget(); // maybe my caller can use that character
        ist.clear(ios_base::failbit); // set the state back to fail()
    }
}
```

Reading a single value

- (At least) three kinds of problems are possible
  - the user types an out-of-range value
  - getting no value (end of file)
  - the user types something of the wrong type (here, not an integer)

```
// first simple and flawed attempt:

cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (cin>>n) {          // read
    if   (1<=n && n<=10) break;   // check range
    cout << "Sorry, "
         << n
         << " is not in the [1:10] range; please try again\n";
}

// use n here
```

Reading a single value

- What do we want to do in those three cases?
  - handle the problem in the code doing the read?
  - throw an exception to let someone else handle the problem (potentially terminating the program)?
  - ignore the problem?

  - Reading a single value
    - Is something we often do many times
    - We want a solution that's very simple to use

Handle everything: What a mess!

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (cin >> n) {
    if (cin) {   // we got an integer; now check it:
        if (1<=n && n<=10) break;
        cout << "Sorry, " << n << " is not in the [1:10] range; please try again\n";
    }
    else if (cin.fail()) {    // we found something that wasn't an integer
        cin.clear(); // we'd like to look at the characters
        cout << "Sorry, that was not a number; please try again\n";
        for (char ch; cin>>ch && !isdigit(ch); )   // throw away non-digits
            /* nothing */ ;
        if (!cin) error("no input");  // we didn't find a digit: give up
        cin.unget(); // put the digit back, so that we can read the number
    }
    else
        throw runtime_error("no input");  // eof or bad: give up
}
// if we get here n is in [1:10]
```

The mess: trying to do everything at once

- Problem: We have all mixed together
  - reading values
  - prompting the user for input
  - writing error messages
  - skipping past “bad” input characters
  - testing the input against a range

- Solution: Split it up into logically separate parts

What do we want?

- What logical parts do we what?
  - **int get_int(int low, int high);** *// read an int in [**low**..**high**] from **cin***
  - **int get_int();** *// read an int from **cin***
    *// so that we can check the range **int***
  - **void skip_to_int()**; *// we found some "garbage" character*
    // so skip until we find an int

- Separate functions that do the logically separate actions

Skip “garbage”

```
void skip_to_int()
{
    if (cin.fail()) {    // we found something that wasn't an integer
        cin.clear(); // we'd like to look at the characters
        for(char ch; cin>>ch; ) { // throw away non-digits
            if (isdigit(ch) || ch=='-') {
                cin.unget(); // put the digit back,
                             // so that we can read the number
                return;
            }
        }
    }
    error("no input");   // eof or bad: give up
}
```

Get (any) integer

```
int get_int()
{
    int n = 0;
    while (true) {
        if (cin >> n) return n;
        cout << "Sorry, that was not a number; please try again\n";
        skip_to_int();
    }
}
```

Get integer in range

```
int get_int(int low, int high)
{
    cout << "Please enter an integer in the range "
        << low << " to " << high << " (inclusive):\n";
    while (true) {
        int n = get_int();
        if (low<=n && n<=high) return n;
        cout << "Sorry, "
            << n << " is not in the [" << low << ':' << high
            << "] range; please try again\n";
    }
}
```

Use

- Problem:
  - The “dialog” is built into the read operations

```
int n = get_int(1,10);
cout << "n: " << n << endl;

int m = get_int(2,300);
cout << "m: " << m << endl;
```

What do we *really* want?

- That's often the really important question
- Ask it repeatedly during software development
- As you learn more about a problem and its solution, your answers improve

```
// parameterize by integer range and “dialog”

int strength = get_int(1, 10,
                "enter strength",
                "Not in range, try again");
cout << "strength: " << strength << endl;

int altitude = get_int(0, 50000,
                "please enter altitude in feet",
                "Not in range, please try again");
cout << "altitude: " << altitude  << "ft. above sea level\n";
```

[Note: Separating the control from the view is even better]

Parameterize

- Incomplete parameterization: **get_int()** still “blabbers”
  - “utility functions” should not produce their own error messages
  - Serious library functions do not produce error messages at all
    - They throw exceptions (possibly containing an error message)

```
int get_int(int low, int high, const string& greeting, const string& sorry)
{
    cout << greeting << ": [" << low << ':' << high << "]\n";
    while (true) {
        int n = get_int();
        if (low<=n && n<=high) return n;
        cout << sorry  << ": [" << low << ':' << high << "]\n";
    }
}
```

User-defined output: operator<<()

- Usually trivial
  - We often use several different ways of outputting a value
    - Tastes for output layout and detail vary

```
ostream& operator<<(ostream& os, const Date& d)
{
    return os << '(' << d.year()
              << ',' << d.month()
              << ',' << d.day() << ')';
}
```

A Note on *this
or, Making the Sausage

```
void do_some_printing(Date d1, Date d2)
{
    cout << d1;           // means operator<<(cout,d1) ;

    cout << d1 << d2;
            // means (cout << d1)  <<  d2;
            // means (operator<<(cout,d1)) <<  d2;
            // means operator<<((operator<<(cout,d1)), d2) ;
}
```

A Note on *this
or, Making the Sausage

- When calling a method, C++ creates a pointer to the method's object called *this
  - So, in `cout << d1;` , *this points to `cout`
  - C++ sees the above code as
    ```
    operator<<(cout, d1);
    ```
  - The method returns *this, which points to `cout`
- Thus when chaining stream operators, e.g., **`cout << d1 << d2;`** , we get:
  - **`(cout << d1) << d2;`**
  - **`(operator<<(cout,d1)) << d2;`** which is **`(cout) << d2;`**

  or, fully nesting the operations
  - **`operator<<((operator<<(cout,d1)), d2);`**

User-defined input: operator>>()

```
istream& operator>>(istream& is, Date& dd)
    // Read date in format: ( year , month , day )
{
    int y, d, m;
    char ch1, ch2, ch3, ch4;
    is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;
    if (!is) return is;  // we didn't get our values, so just leave
    if (ch1!='(' || ch2!=',' || ch3!=',' || ch4!=')') { // oops: format error
        is.clear(ios_base::failbit);     // something wrong: set state to fail()
        return is;           // and leave
    }
    dd = Date{y,Month(m),d};  // update dd
    return is;            // and leave with is in the good() state
}
```

What We Learned Today

- More inheritance, 1 of 3 pillars of OOP
  - Practiced simple inheritance, with a sneak peak at polymorphism
  - Discussed multiple inheritance and the diamond problem
  - Looked at (typical) class memory layouts
  - Introduced pure virtual functions
- File Input / Output
  - Introduced the stream model of I/O (shared with bash)
  - Covered file definition, and how to read and write them using streams
  - Discussed the 4 stream states – good, bad, fail, and eof
  - Examined some code organization issues with files
  - Considered what underlies “*this” pointer to the current object

Quick Review

- What is multiple inheritance?
- How does UML model multiple inheritance?
- How is multiple inheritance specified in C++?
- Explain the “Diamond Problem”.
  - How does explicit base class method calls help?
  - How does virtual inheritance help?
- True or False: The layout of C++ objects in memory is defined in the language standard.

Quick Review

- A named sequence of bytes available via the operating system is called a _____.
- To parse a file, we must know its _____ and ____ _____.
- Throwing _____ reports an generic error that occurred during the time the program was running. The parameter is a _____ that describes the error.
- When reading from a stream, what do these statuses mean?
  - good() – eof () – bad() – fail()
- The end of file keystroke is ^__ in Windows and ^__ in Mac / Linux.
- The _____ method of the stream allows us to specify a mask enabling the throwing of exceptions for desired stream events.

For Next Class

- (Optional) Read Chapter 10 in Stroustrop
  - Do the Drills!
- Skim Chapter 11 for next lecture
  - Formatted I/O using streams
  - UML Activity Diagrams
  - Decorator Pattern

Homework #3 Questions?

This assignment involves writing a word / phrase guessing game (always a good way to learn a new language and new technology). We'll utilize a Makefile, use either the ddd or gdb debugger, employ git (of course!), and use Umbrello to design our class and use case diagrams.

Due Thursday, February 8 at 8 am.