CSE 1325: Object-Oriented Programming

Lecture 17 – 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

Quick Review

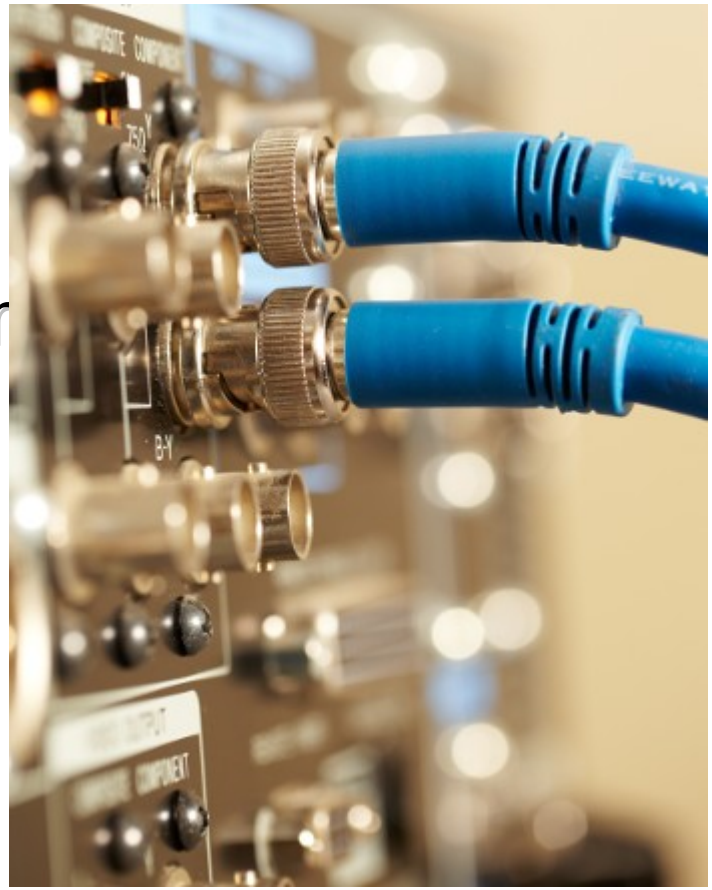
- True or **False**: A class (type) may not be used to instantiate templates such as vector
- **True** or False: The only difference in C++ between a struct and a class is the default visibility of its members.
- Ensuring that a program operates on clean, correct and useful data is **data validation**.
- Algorithmically enforceable constraints on the correctness, meaningfulness, and security of input data is **validation rules**.
- Which of the following help with that:(1) Clever programming
  (2) Invariant classes (3) Validation methods (4) Validation in the constructor
  (5) Limiting members to primitives
- The following are good reasons to make class fields private:
  (1) To provide a clean interface (2) To hide proprietary data
  (3) To maintain an invariant (4) To enforce HIPPA regulations
  (5) To ease debugging
  (6) To change representation or algorithm without impacts

Quick Review

- Providing a user-defined meaning to a pre-defined operator (e.g., +, ==, <<) for a user-defined type (class) is **operator overloading**.
- Which of the following operators can NOT be overloaded: **(1) ::** (2) new **(3) ?:** (4) && (This will NOT be on the exam!)
- To overload the + operator for a class, define the **operator+** method.
- 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** (or **class diagram**) 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?
  “public” maintains the same visibility of inherited members.
  “protected” changes inherited public members to protected.
  “private” changes inherited public and protected members to private.

Overview

- Inheritance / Polymorphism 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 Inheritance and Polymorphism

```
#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;
};
int main() {
  vector<Critter*> critters;

  critters.push_back(new Critter{13});
  critters.push_back(new Critter{11});
  critters.push_back(new Critter{7});
  critters.push_back(new 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));
  }
}
```

```
ricegf@pluto:~/dev/cpp/201701/17$ make v1
g++ -std=c++11 -o v1 critter_v1.cpp
ricegf@pluto:~/dev/cpp/201701/17$ ./v1
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!
Generic critter sound!
```

Without
Inheritance and Polymorphism

```
#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;
};
int main() {
  vector<Critter*> critters;

  critters.push_back(new Critter{13});
  critters.push_back(new Critter{11});
  critters.push_back(new Critter{7});
  critters.push_back(new 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...)?

With Inheritance and Polymorphism

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

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

With Inheritance and Polymorphism

```
int main() {
  vector<Critter*> critters;

  critters.push_back(new Critter{13});
  critters.push_back(new Cow{11});
  critters.push_back(new Dog{7});
  critters.push_back(new Chicken{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...)?

Literally 4 words changed!
Added 3 derived classes – but mostly just the portions that differed from the base class.

```
ricegf@pluto:~/dev/cpp/201701/17$ make v2
g++ -std=c++11 -o v2 critter_v2.cpp
ricegf@pluto:~/dev/cpp/201701/17$ ./v2
W E L C O M E   T O   T H E   B A R N Y A R D !
Generic critter sound!
Moo! Mooooo!
Woof! Woof!
Cluck! Cluck!
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
Moo! Mooooo!
Cluck! Cluck!
Generic critter sound!
Woof! Woof!
Cluck! Cluck!
```

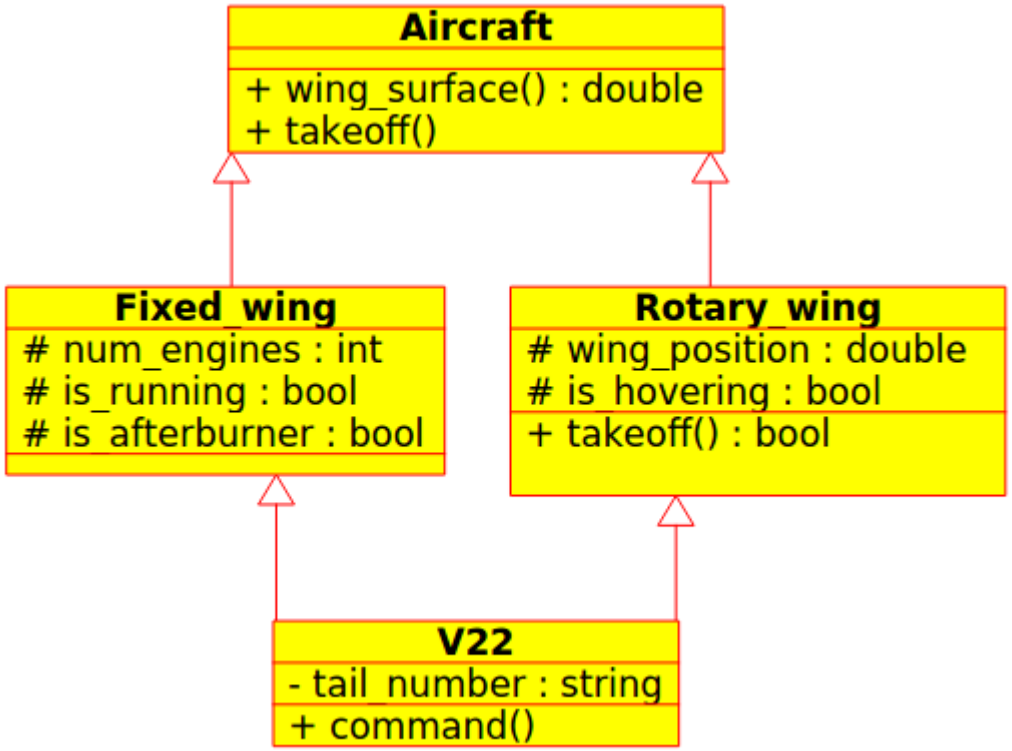
Multiple Inheritance

- What if a class is derived from more than one base class?
  - This is called *multiple inheritance*
  - MI is often helpful, e.g., with the Adapter Pattern
    - Given the Client and Server classes, simply inherit both!
- 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.

Multiple Inheritance

Example with Multiple Inheritance



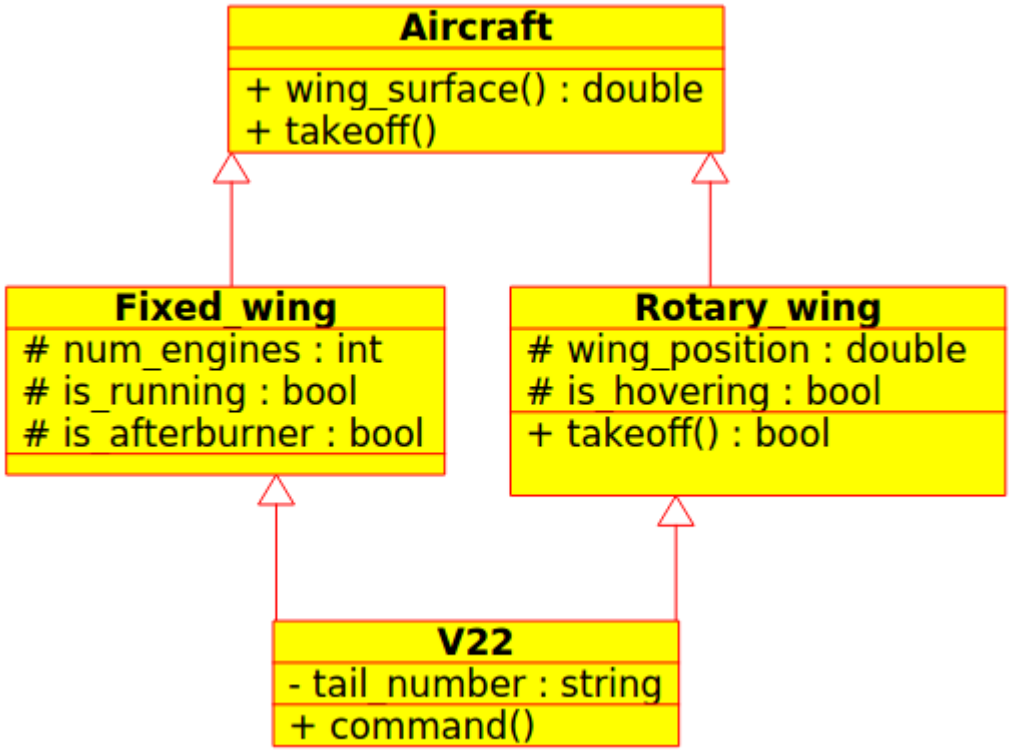
V22 would have wing_surface(), takeoff(), and command() methods

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?

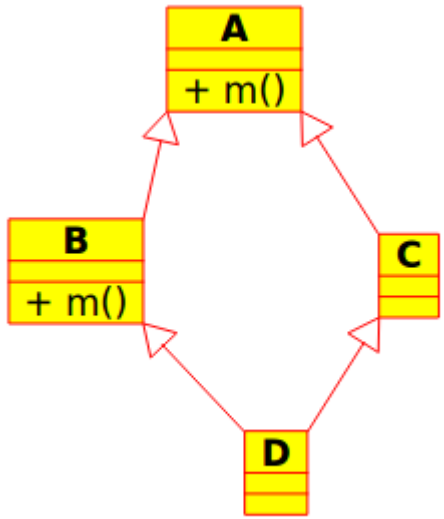
https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem
http://www.cprogramming.com/tutorial/multiple_inheritance.html
http://www.cprogramming.com/tutorial/virtual_inheritance.html

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.

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




```
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$
```

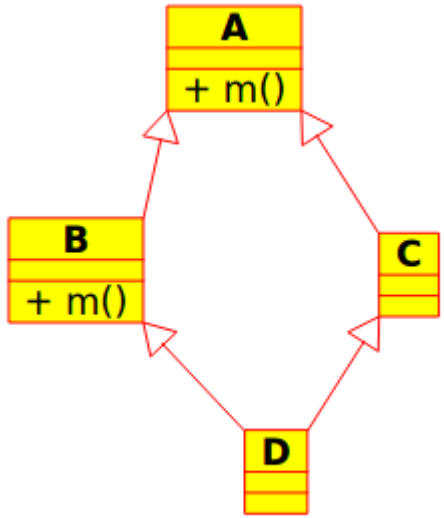
Explicit Base Class Method Call

```
#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$
```

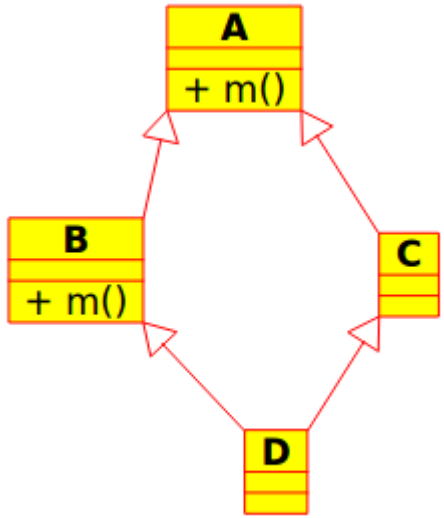
What if we specify the method using namespace notation?



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...

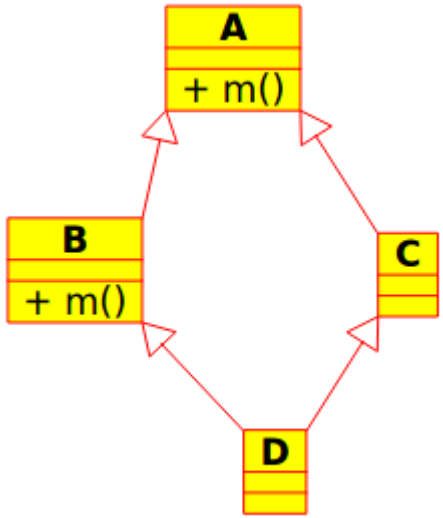


Uh oh

```
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$
```

This is a Memory Layout Issue

- B adds A's method to the end of its memory block
- C adds A's method to the end of its memory block
- D now has two A's – the one in B and the one in C – and doesn't know which A you mean
- Proposed solution: Use pointers and only keep a single A!



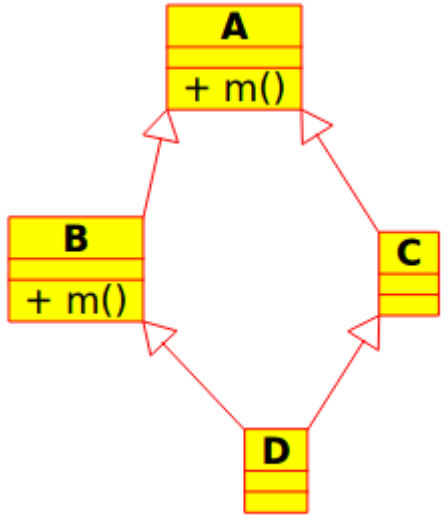
Memory Layout

Memory for D
Memory for B
method m
Memory for A
method m
Memory for C
method m
Memory for A
method m

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();
}
```

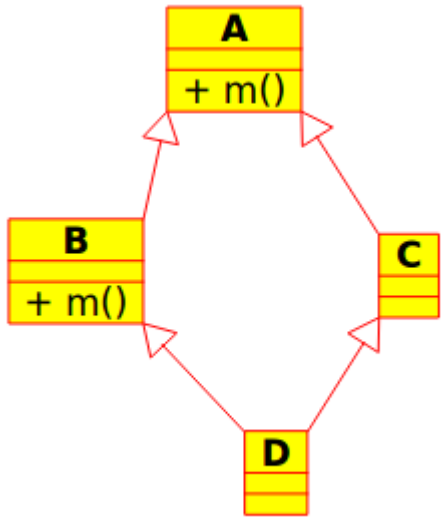
Easier done than said.
Just make the intermediate classes inherit *virtually*.

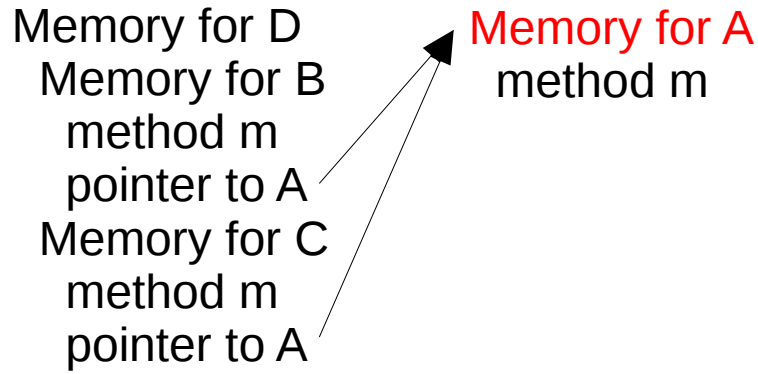



```
ricegf@pluto:~/dev/cpp/201701/17$ g++ -std=c++11 diamond.cpp
ricegf@pluto:~/dev/cpp/201701/17$ ./a.out
m of A
m of B
m of A
ricegf@pluto:~/dev/cpp/201701/17$
```

How Does Virtual Inheritance Work?

- By keeping a single block of memory allocated for A, the ambiguity is resolved





Object Layout in Memory

```
class Parent1 {
public:
    int a1, b1;
    virtual void foo()
      {cout << "parent 1" << endl;}
};
class Parent2 {
public:
    int a2, b2;
    virtual void foo()
      {cout << "parent 2" << endl;}
};
class Child : public Parent1,
              public Parent2 {
public:
    int c, d;
    virtual void foo()
      {cout << "child" << endl;}
};
int main(){
    Parent1 p1;
    Parent2 p2;
    Child c;

    p1.foo();
    p2.foo();
    c.foo();
```

```
    cout << "Parent1 Offset a1 = "
      << (size_t)&p1.a1 - (size_t)&p1 << endl;
    cout << "Parent1 Offset b1 = "
      << (size_t)&p1.b1 - (size_t)&p1 << endl;

    cout << "Parent2 Offset a2 = "
      << (size_t)&p2.a2 - (size_t)&p2 << endl;
    cout << "Parent2 Offset b2 = "
      << (size_t)&p2.b2 - (size_t)&p2 << endl;

    cout << "Child Offset a1 = "
      << (size_t)&c.a1 - (size_t)&c << endl;
    cout << "Child Offset b1 = "
      << (size_t)&c.b1 - (size_t)&c << endl;
    cout << "Child Offset a2 = "
      << (size_t)&c.a2 - (size_t)&c << endl;
    cout << "Child Offset b2 = "
      << (size_t)&c.b2 - (size_t)&c << endl;
    cout << "Child Offset c = "
      << (size_t)&c.c - (size_t)&c << endl;
    cout << "Child Offset d = "
      << (size_t)&c.d - (size_t)&c << endl;
}
```

Object Layout in Memory

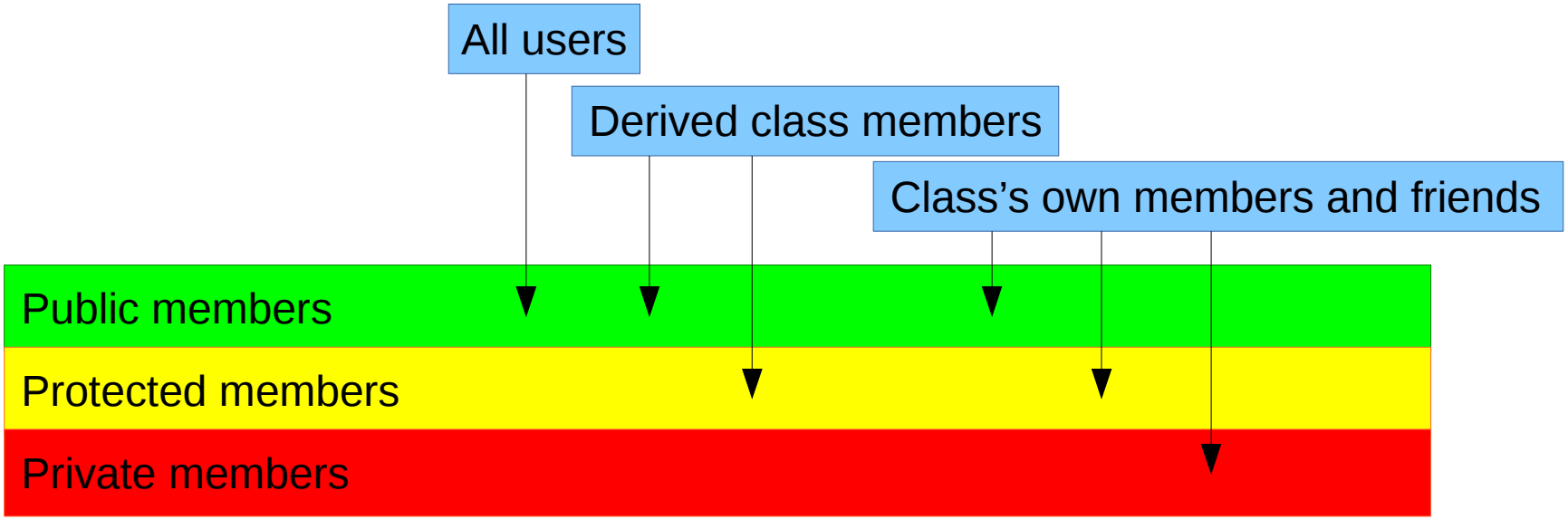
```
class Parent1 {
public:
    int a1, b1;
    virtual void foo()
      {cout << "parent 1" << endl;}
};
class Parent2 {
public:
    int a2, b2;
    virtual void foo()
      {cout << "parent 2" << endl;}
};
class Child : public Parent1,
public:
    int c,
    virtual
      {cout
};
int main(){
    Parent1
    Parent2
    Child c

    p1.foo(
    p2.foo(
    c.foo()
```

```
ricegf@pluto:~/dev/cpp/19$ ./a.out
parent 1
parent 2
child
Parent1 Offset a1 = 8
Parent1 Offset b1 = 12
Parent2 Offset a2 = 8
Parent2 Offset b2 = 12
Child Offset a1 = 8
Child Offset b1 = 12
Child Offset a2 = 24
Child Offset b2 = 28
Child Offset c = 32
Child Offset d = 36
ricegf@pluto:~/dev/cpp/19$
```

```
    cout << "Parent1 Offset a1 = "
      << (size_t)&p1.a1 - (size_t)&p1 << endl;
    cout << "Parent1 Offset b1 = "
      << (size_t)&p1.b1 - (size_t)&p1 << endl;

    cout << "Parent2 Offset a2 = "
      << (size_t)&p2.a2 - (size_t)&p2 << endl;
    cout << "Parent2 Offset b2 = "
      << (size_t)&p2.b2 - (size_t)&p2 << endl;

    cout << "Child Offset a1 = "
      << (size_t)&c.a1 - (size_t)&c << endl;
    cout << "Child Offset b1 = "
      << (size_t)&c.b1 - (size_t)&c << endl;
    cout << "Child Offset a2 = "
      << (size_t)&c.a2 - (size_t)&c << endl;
    cout << "Child Offset b2 = "
      << (size_t)&c.b2 - (size_t)&c << endl;
    cout << "Child Offset c = "
      << (size_t)&c.c - (size_t)&c << endl;
    cout << "Child Offset d = "
      << (size
```

BUT
Memory layout of objects is not specified in the standard, but is *compiler-specific!*

Access model



- A member (data, function, or type member) or a base can be
  - Private, protected, or public

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")

```
class Engine {       // interface to electric motors
    // no data
    // (usually) no constructor
    virtual double increase(int i) =0;       // must be defined in a derived class
    // …
    virtual ~Engine();       // (usually) a virtual destructor
};
Engine eee;       // error: Collection is an abstract class
```

Pure virtual functions

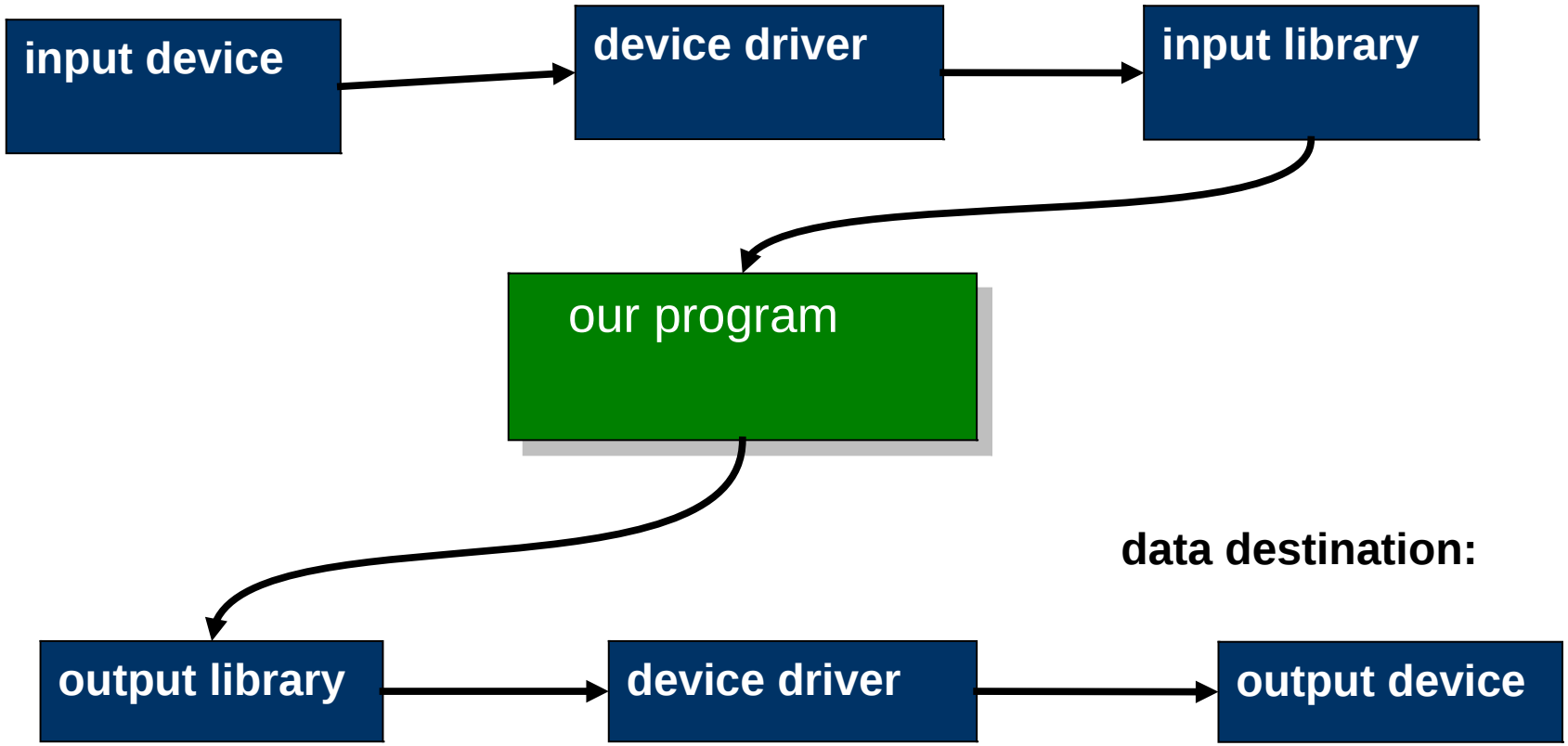
- A pure interface can then be used as a base class

```
Class M123 : public Engine {        // engine model M123
   // representation
public:
   M123();   // construtor:  initialization, acquire resources
   double increase(int i) { /* … */ }  // overrides Engine ::increase
   // …
   ~M123(); // destructor: cleanup, release resources
};

M123 window3_control;     // OK
```

Switching Gears Now
To I/O and Files

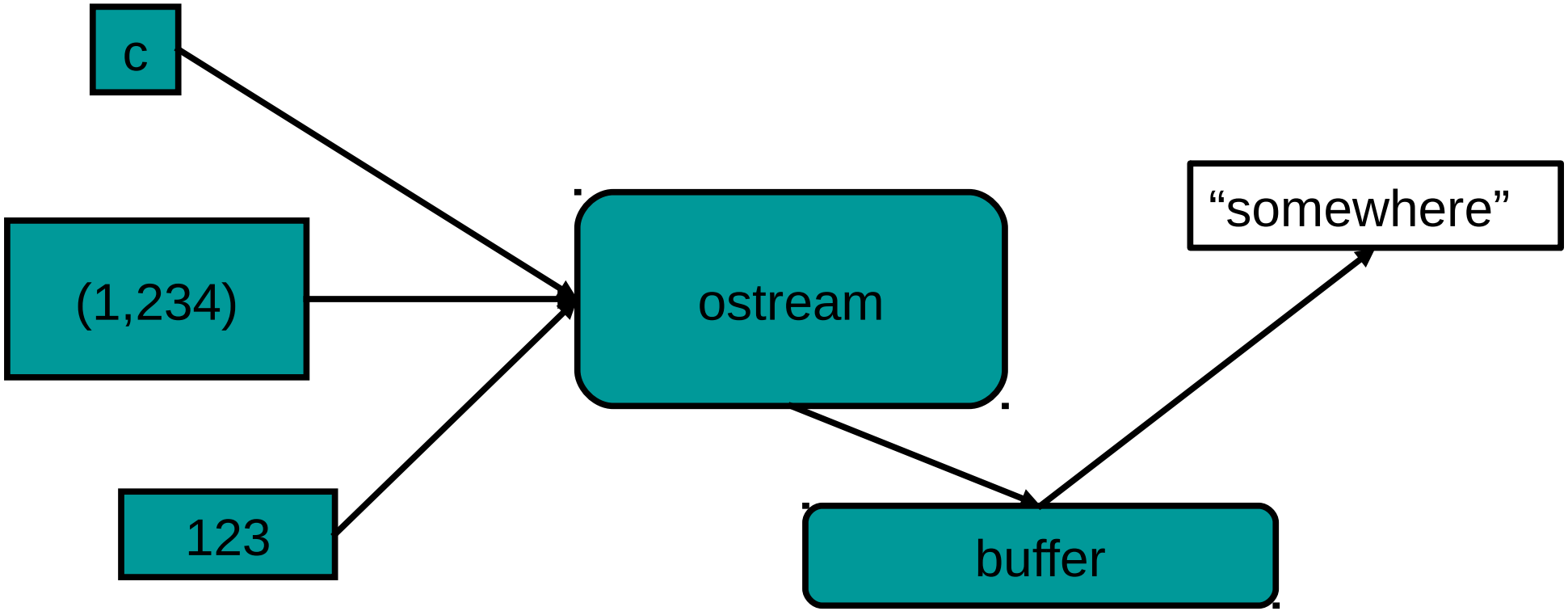
Input and Output

data source:



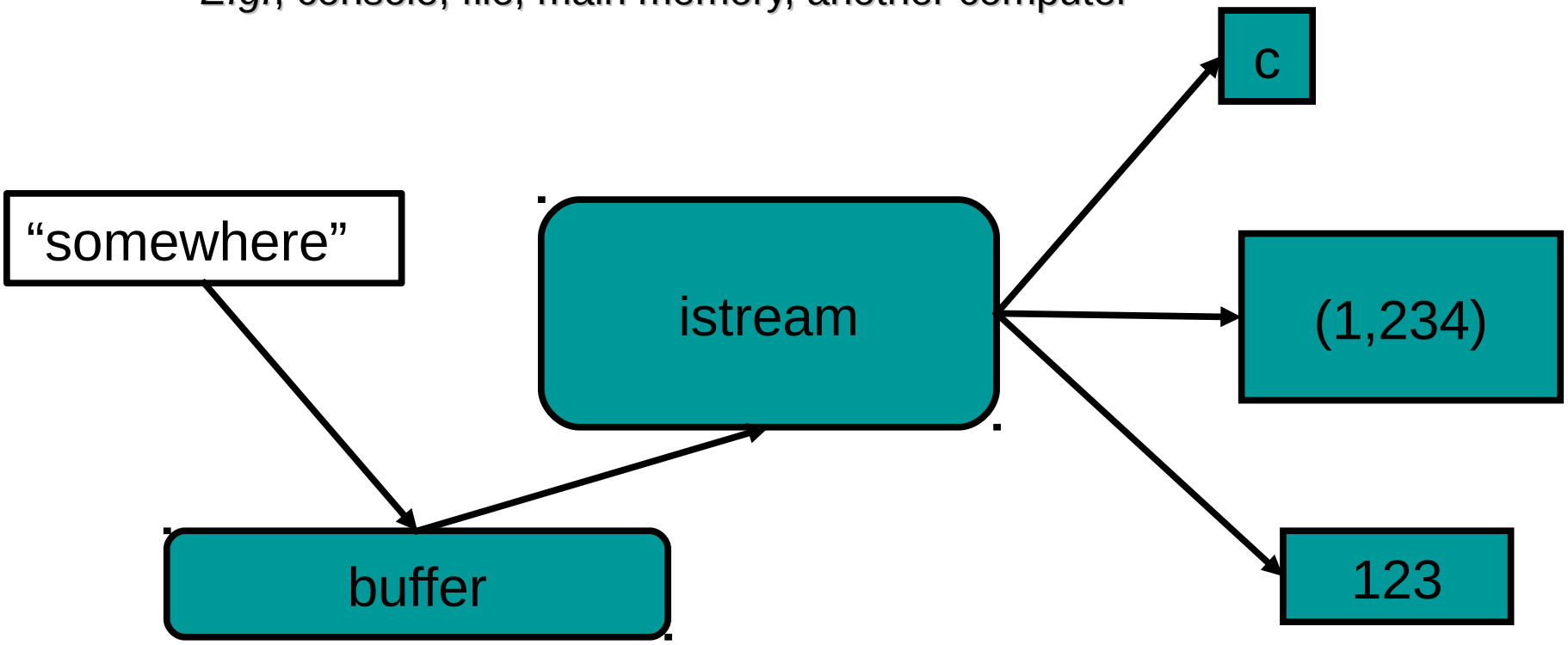
The stream model

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

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

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 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

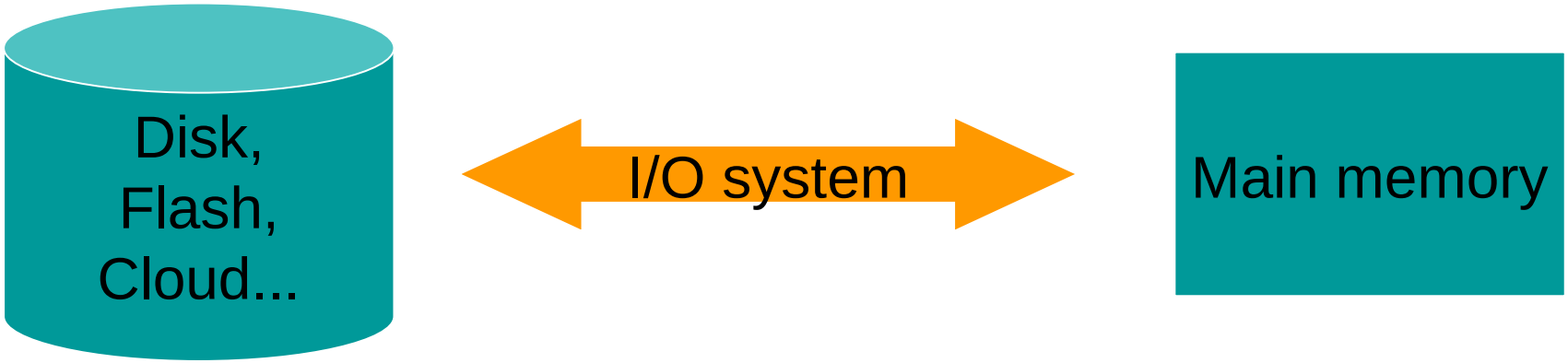
- At the fundamental level, a file is a sequence of bytes numbered from 0 upwards
- Other notions can be supplied by programs that interpret a “file format”
  - For example, the 6 bytes "123.45" might be interpreted as the floating-point number 123.45

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

```
#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);
    // …
```

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

Opening a file for writing

```
#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
}
```

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
- **Sprint #2 now in progress: It's GUI Week!**
  - **Individuals**: Create your main window, and dialogs for instancing a serving (and a serving class)
  - **Duos:** Also create dialogs for instancing servers and customers, and to create orders (and an order class)
  - **Trios:** Also manage the state of the orders, and create the Emporium itself (and an emporium class)
  - **Quattros:** Also display orders for servers and customers, and load and save data to files