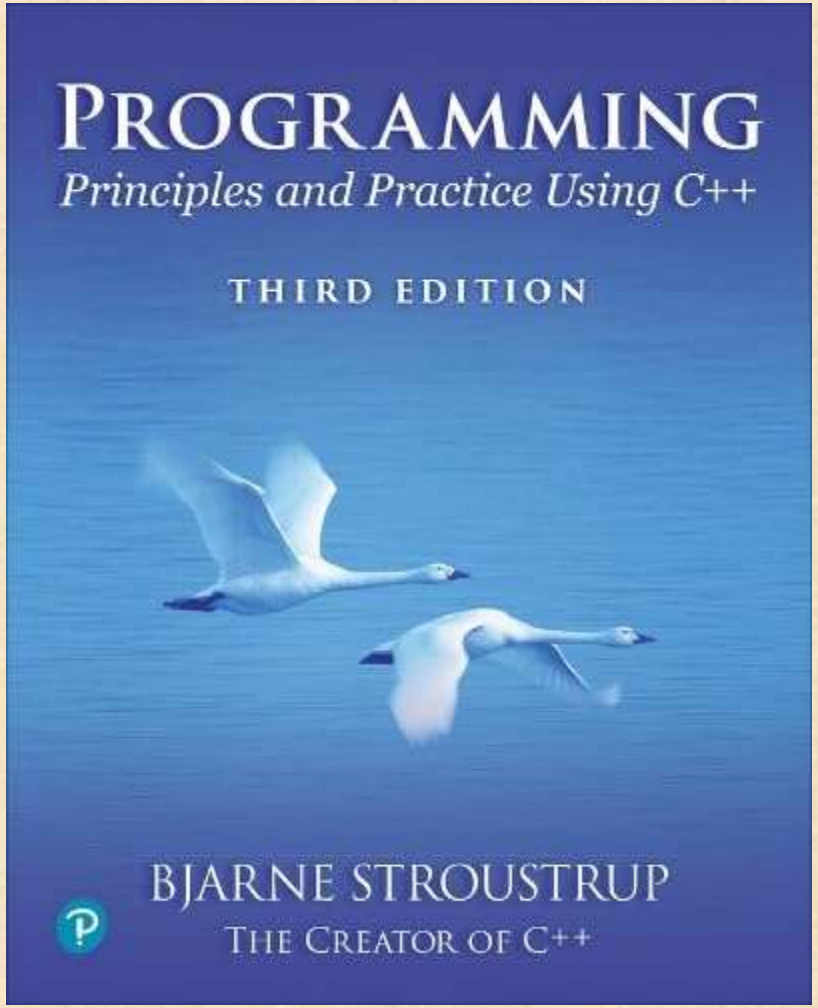
Chapter 9 – Input and Output Streams

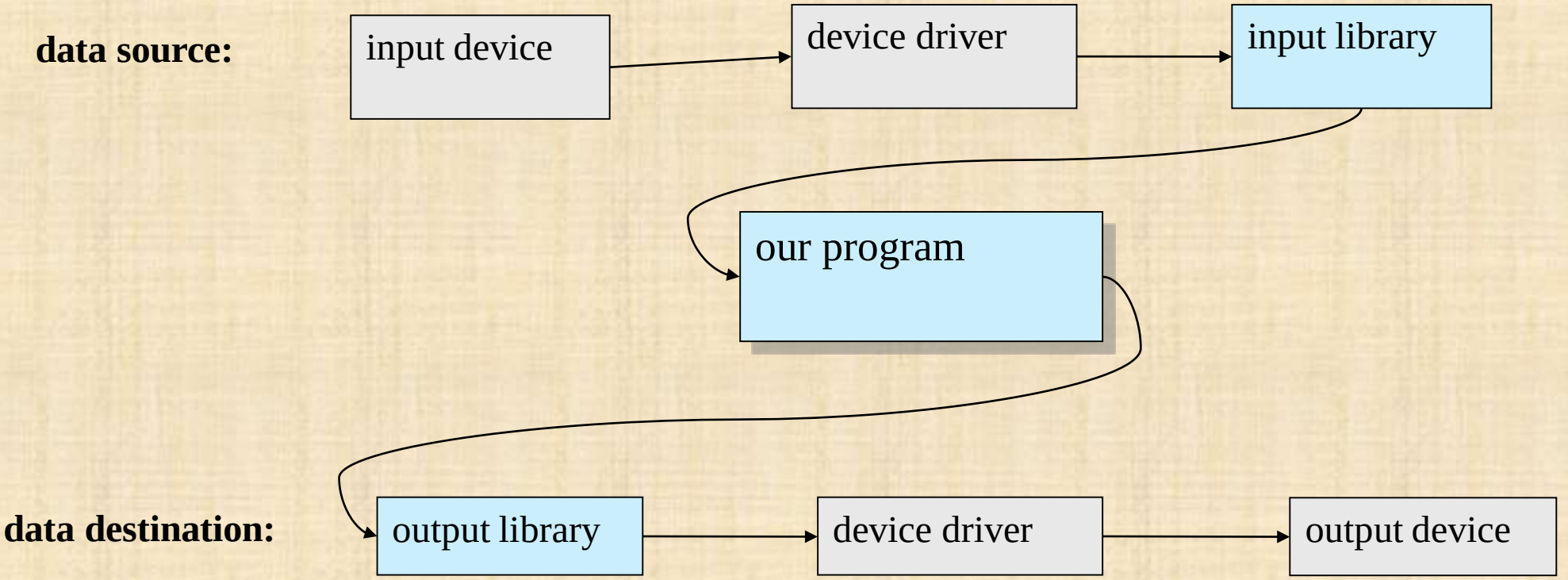


Science is what we have learned about
how to keep from fooling ourselves.
– Richard P. Feynman

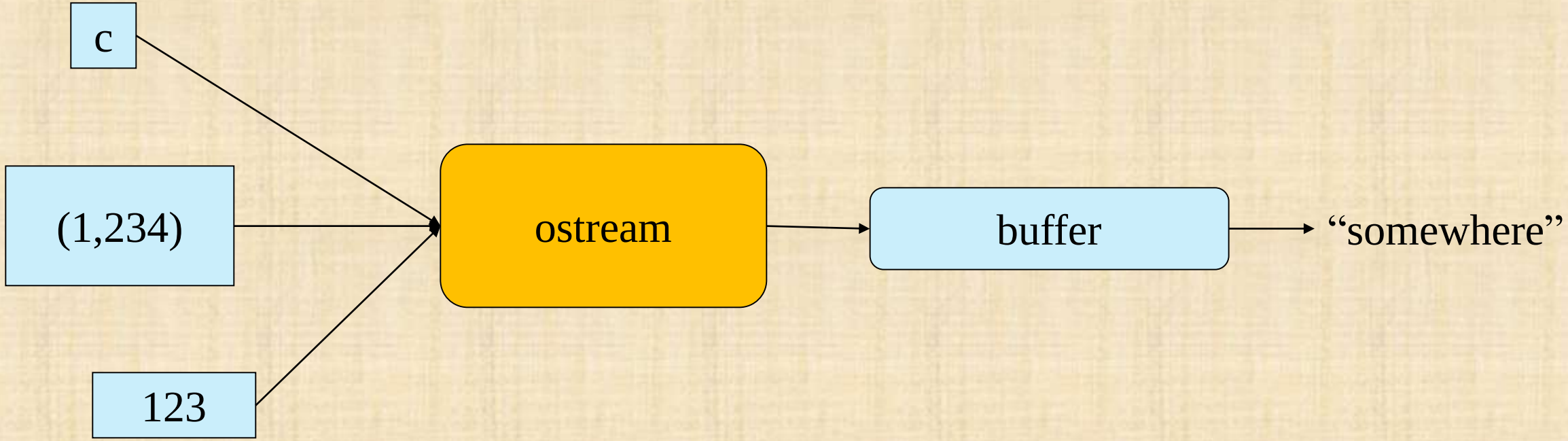
Overview

- Input and output
- The I/O stream model
- Files
  - Opening a file; Reading and writing a file
- I/O error handling
- Reading a single value
  - Breaking the problem into manageable parts; Separating dialog from function
- User-defined I/O
- Formatting
- Character I/O
- String streams

Input and Output

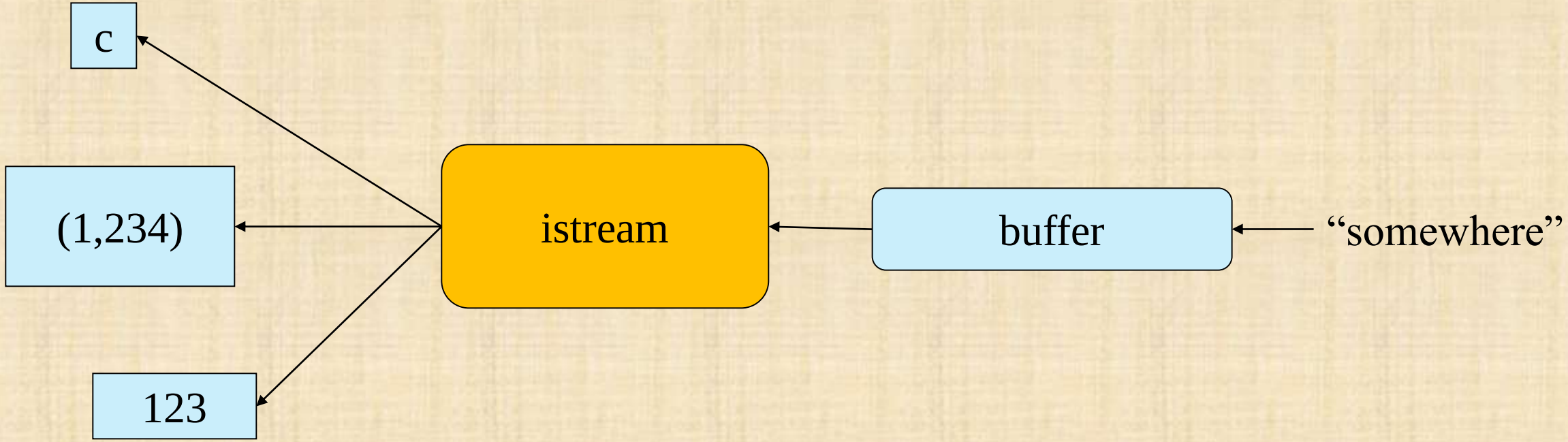


The stream model



- 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

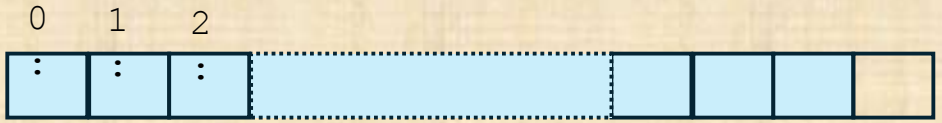
The stream model

- 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
  - Extensible
    - You can define your own I/O operations for your own types
  - Stream oriented
    - A stream can be attached to any I/O or storage device
    - Some format properties can be set for all values sent to a stream

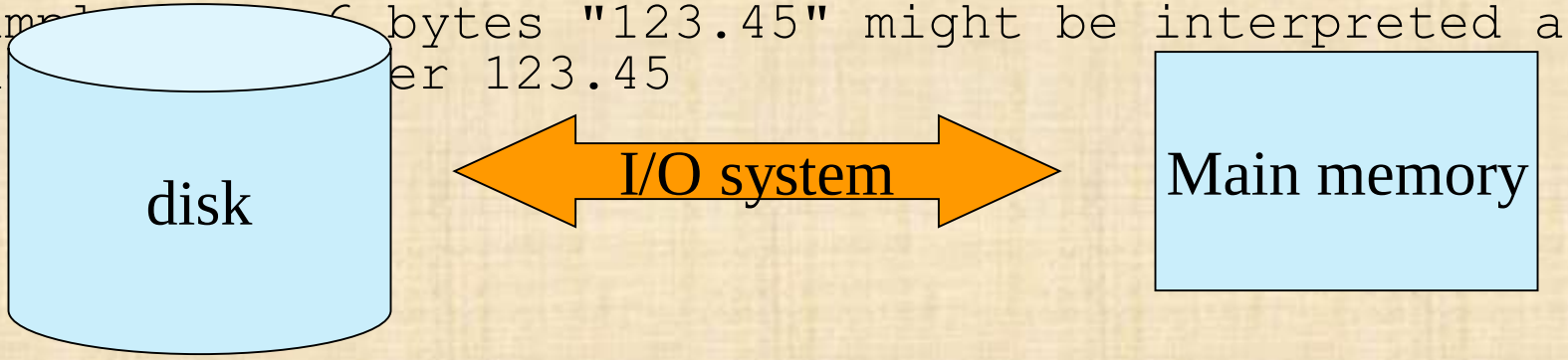
Files

- 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 sequence of bytes stored in permanent storage
  - A file has a name
  - The data on a file has a format
- We can read/write a file if we know its name and format

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, 6 bytes "123.45" might be interpreted as the floating point number 123.45



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)
    - An attempt to open a file may fail
  - Then we can read
  - Then we must close it
    - That is typically done implicitly
- To write a file
  - We must name it
  - We must open it (for writing)
    - Or create a new file of that name
    - An attempt to open a file may fail
  - Then we can write it
  - We must close it
    - That is typically done implicitly

Opening a file for reading

- To read a file
  - We must know its name
  - We must open it (for reading)
    - An attempt to open a file may fail
  - Then we can read
  - Then we must close it
    - That is typically done implicitly

```
cout << "Please enter input file name: ";
string iname;
cin >> iname;
ifstream ist {iname};  // ifs is an input stream from named iname
if (!ist)
     error("can't open input file ", iname);
```

Opening a file for writing

- To write a file
  - We must name it
  - We must open it (for writing)
    - Or create a new file of that name
    - An attempt to open a file may fail
  - Then we can write it
  - We must close it
    - That is typically done implicitly

```
cout << "Please enter name of output file: ";
string oname;
cin >> oname;
ofstream ofs {oname};   // ofs is an output stream from
  a file name oname
if (!ofs)
 error("can't open output file ", oname);
```

Streams have destructors

- Use constrcutors and destructors to control lifetime

```
int main()
{
    string iname;
    // …
    istream ist {iname};
    // …
    string oname;
    // …
    ostream {oname};
    // …
} // here, files are implicitly closed, I/O buffers and strings are given back to the free store
```

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

```
struct Reading { // a temperature reading
     int hour;   // hour after midnight [0:23]
     double temperature;
};

vector<Reading> temps; // create a vector to store the readings

int hour;
double temperature;
while (ist >> hour >> temperature) {                    // read
     if (hour < 0 || 23 <hour)                          // check (always
check validity of input)
          error("hour out of range");
     temps.push_back( Reading{hour,temperature} );     // 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*
  - **bad()** // *something unexpected and serious happened*

Sample integer read "failure"

- Ended by "terminator character" – program to deal with that
  - 1 2 3 4 5 *
  - State is **fail()**
- Ended by format error – program to deal with that
  - 1 2 3 4 5.6
  - State is **fail()**
- Ended by "end of file" – often the desired end
  - 1 2 3 4 5 end of file
  - 1 2 3 4 5 Control-Z (Windows)
  - 1 2 3 4 5 Control-D (Unix)
  - State is **eof()**
- Something really bad – you usually can't recover from that
  - Disk format error
  - State is **bad()**

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()) 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
```

Throw an exception for bad()

- Often, we cannot recover from **bad**
  - E.g., disk read error, unhandled network failure

```
// How to make ist throw if it goes bad:
ist.exceptions(ist.exceptions()|ios_base::badbit);

// can be read as
//     "set ist's exception mask to whatever it was plus badbit"
//     or as "throw an exception if the stream goes bad"
```

Given that, we can simplify our input loop by no longer checking for **bad**

Simplified input loop

```
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
```

Throw an exception for bad()

- Often, we don't want to try to recover from **fail**
  - E.g., file-read format error

```
ist.exceptions(ist.exceptions()|ios_base::badbit
 ()|ios_base::failbit);
```

Given that, we can simplify our input loop to the simple and obvious:

```
void fill_vector(istream& ist, vector<int>& v)
    // read integers from ist into v until we reach eof()
{
    for (int x; ist>>x; )
          v.push_back(x);
}
```

Reading a single value

```
// 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
```

- 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

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); )  /* nothing */ ;    //
  throw away non-digits                  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
```

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

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

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

- Problem:
  - The "dialog" is (still) built into the read operations

What do we ***really*** want?

```
// 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";
```

- 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

```
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";
 }
}
```

- 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)

User-defined output: operator<<()

- Usually trivial

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

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

Use

```
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)
 ;
}
```

User-defined input: operator>>()

- // Input is usually messier than input: formatting and errors

```
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)                                    // we didn't get our values, so just leave
        return is;
 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
```

Character I/O

- Sometimes, we have to drop down a level and read and look at individual characters
  - E.g., to tokenize **1+4*x<=y/z*5**

```
for (char ch; cin.get(ch); ) {          // get(ch) reads the next character into ch; doesn't skip whitespace
        if (isspace(ch)) {
                // do nothing; i.e., skip whitespace (e.g. space or tab)
        }
        else if (isdigit(ch)) {
                // .. read a number ...
        }
        else if (isalpha(ch)) {
                // ... read an identifier ...
        }
        else {
                // ... deal with operators ...
        }
}
```

Character classification

- Useful (and fast) standard-library functions for dealing with characters

Character classification	
isspace(c)	Is c whitespace (' ', '\t', '\n', etc.)?
isalpha(c)	Is c a letter ('a'..'z', 'A'..'Z') (note: not '_')?
isdigit(c)	Is c a decimal digit ('0'..'9')?
isxdigit(c)	Is c a hexadecimal digit (decimal digit or 'a'..'f' or 'A'..'F')?
isupper(c)	Is c an uppercase letter?
islower(c)	Is c a lowercase letter?
isalnum(c)	Is c a letter or a decimal digit?
iscntrl(c)	Is c a control character (ASCII 0..31 and 127)?
ispunct(c)	Is c not a letter, digit, whitespace, or invisible control character?
isprint(c)	Is c printable (ASCII ' '..'~')?
isgraph(c)	Is isalpha(c) or isdigit(c) or ispunct(c) (note: not space)?

Character case	
x=toupper(c)	x becomes c or c's uppercase equivalent
x=tolower(c)	x becomes c or c's lowercase equivalent

String streams – using strings (in memory) as I/O sources and targets

```
Point get_coordinates(const string& s)        // extract {x,y} from "(x,y)"
{
        istringstream is {s}; // make a stream so that we can read from s
        Point xy;
        char left_paren, ch, right_paren;
        is >> left_paren >> xy.x >> ch >> xy.y >> right_paren;
        if (!is || left_paren !='(' || ch!=',' || right_paren!=')')
                error("format error: ",s);
        return xy;
}
auto c1 = get_coordinates("(2,3)");
auto c2 = get_coordinates("( 200, 300) ");
auto c3 = get_coordinates("100,400");         // will call error()
```

Formatting

- An integer can be printed in different bases
  - Decimal, hexadecimal (base 16), octal (base 8), and binary (base 2)
- A floating-point value can be printed in one of 4 formats
  - **fixed** (e.g., **12.67**), scientific (**e.g., 1.234e7**),
  - **defaultfloat** (what fits best in a field), **hexfloat** (hexadecimal mantissa and exponent)
- You can choose the **precision** used for writing a number
- You can choose the **width** of a field for a number (the number of character positions used)
- You can choose the placement of a number within a field (**adjustfield**) (**left, right, internal**)

- See PPP3§9.10 for some details, and cppreference for many details

Formatting

- **printf()** is arguably that most popular C function, and a reason for C's success
- But it is not type safe

```
printf("an int %g and a string '%s'\n","Hello!",123);     // oops!
```

- Nor is it easily extensible

```
printf("a Point %P\N", Point{100,200});     // oops!
```

- C++ now have a printf()-like format() without the opportunities for type errors

```
cout << format("an int {} and a string '{}'\n","Hello!",123);   // print the arguments according to their type
```

  - And a (less obvious) mechanism for extensibility

```
cout << format("a Point %P\n", Point{100,200});          // works if you have defined %P to refer to Point
```

- See PPP3§9.10.6 for some details, and cppreference for many details

Next Lecture

Graphical output