Parasol
Smarter computing.
Texas A&M University

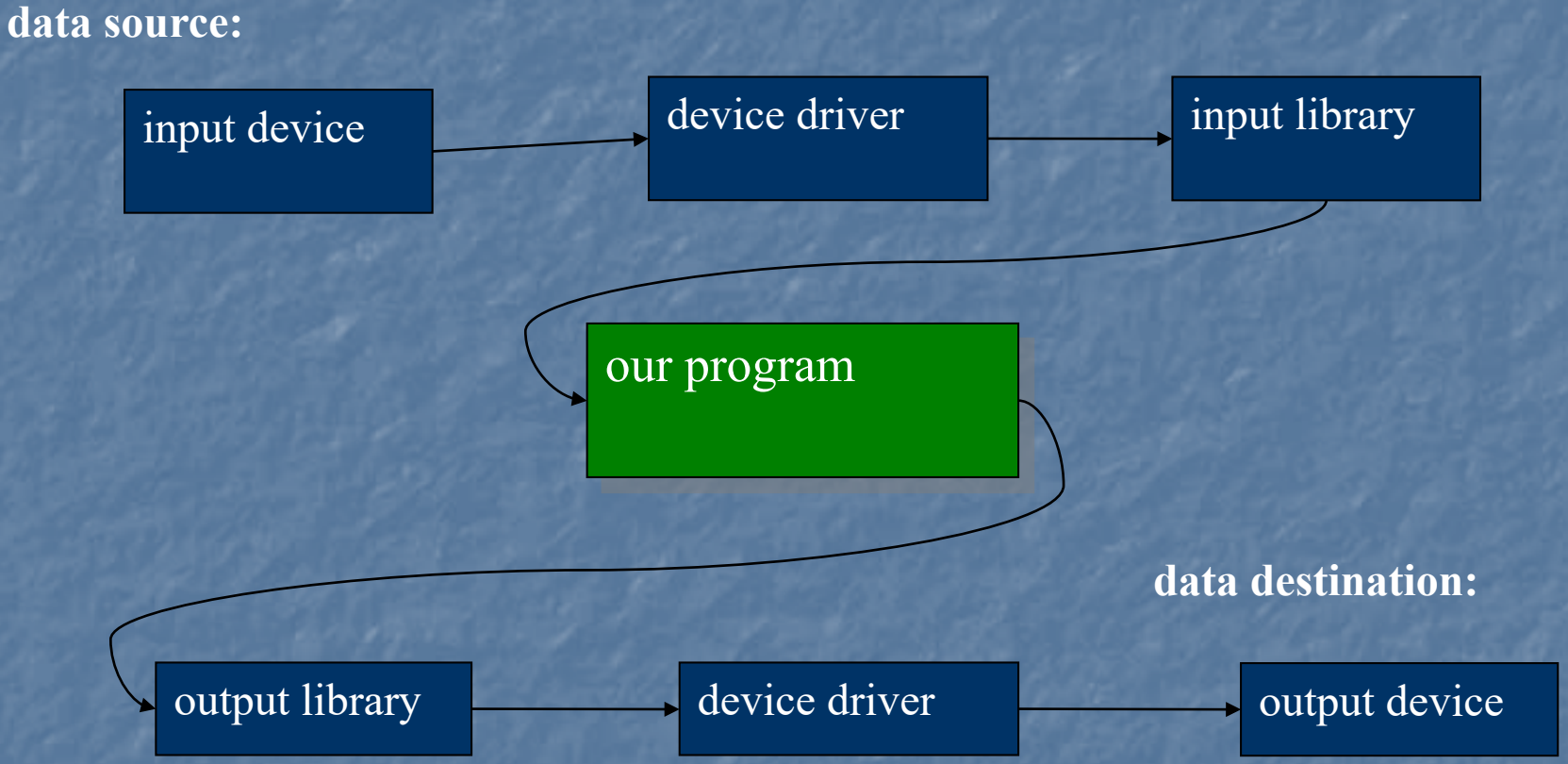
# Chapter 10
# Input/Output Streams

Bjarne Stroustrup
www.stroustrup.com/Programming

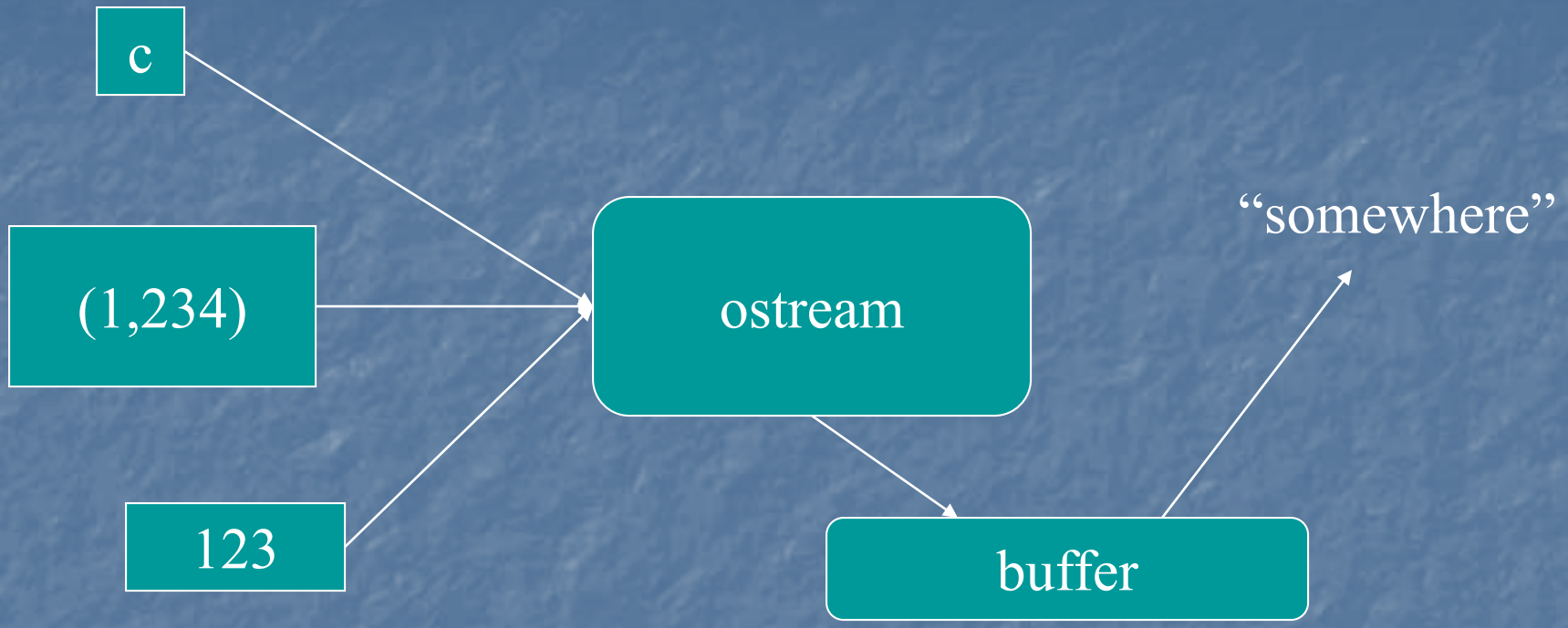
# Overview

- Fundamental I/O concepts
- Files
  - Opening
  - Reading and writing streams
- I/O errors
- Reading a single integer

# Input and Output

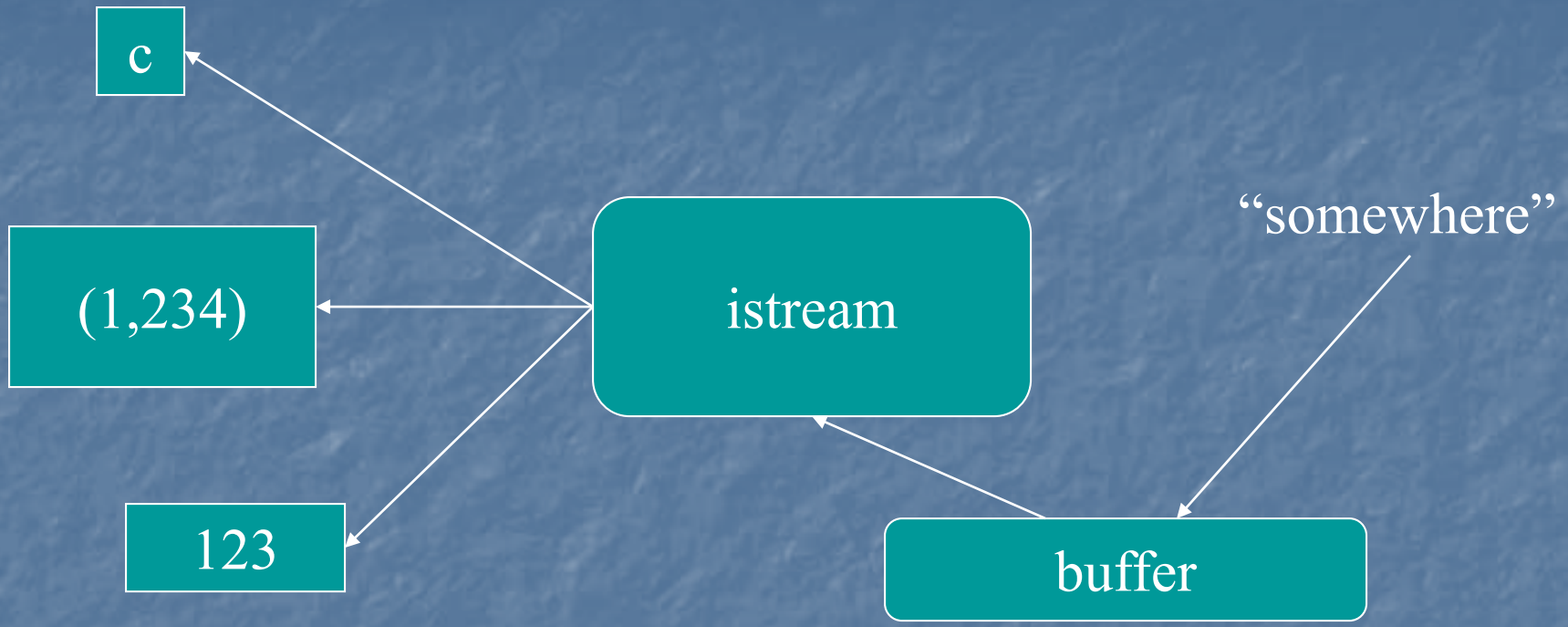


# The stream model




- 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

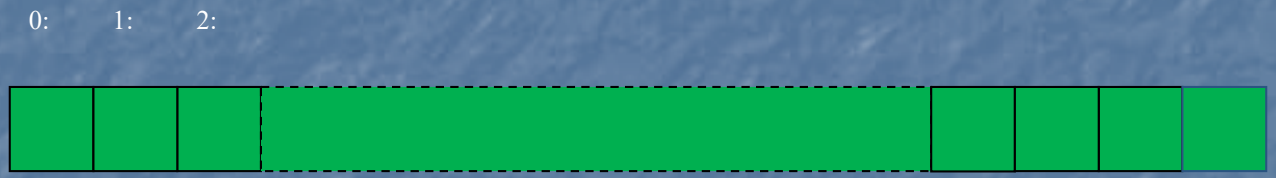


- Reading and writing
  - Of typed entities
    - << (output) and >> (input) plus other operations
    - Type safe
    - Formatted
  - Typically stored (entered, printed, etc.) as text
    - But not necessarily (see binary streams in chapter 11)
  - Extensible
    - You can define your own I/O operations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

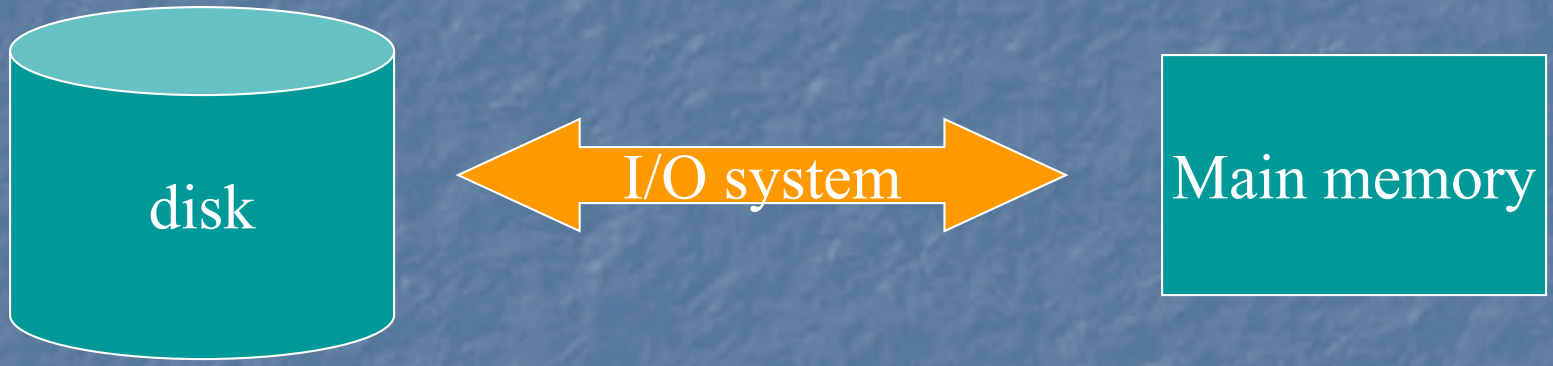
# A file




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

# 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 name it
  - 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

Parasol
Smarter computing.
Texas A&M University

```
// …
int main()
{
    cout << "Please enter input file name: ";
    string iname;
    cin >> iname;
    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) error("can't open input file ", iname);
    // …
```

# Opening a file for writing

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

# 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

Parasol
Smarter computing.
Texas A&M University

```
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) error("hour out of range");   // check
        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"
  - 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 (Unix)
  - State is **eof()**
- Something really bad
  - 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 back to fail()
            }
    }
}
```

# Throw an exception for bad()

Parasol
Smarter computing.
Texas A&M University

*// 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 loops by no longer checking for **bad**

# Simplified input loop

Parasol
Smarter computing.
Texas A&M University

```
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

Parasol
Smarter computing.
Texas A&M University

```
// 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 an integer)

# 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
        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?

Parasol
Smarter computing.
Texas A&M University

- 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 built into the read operations

# What do we *really* want?

Parasol
Smarter computing.
Texas A&M University

```
// 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>>()

```
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
}
```

Parasol
Smarter computing.
Texas A&M University

# Next Lecture

**Customizing input and output (chapter 11)**