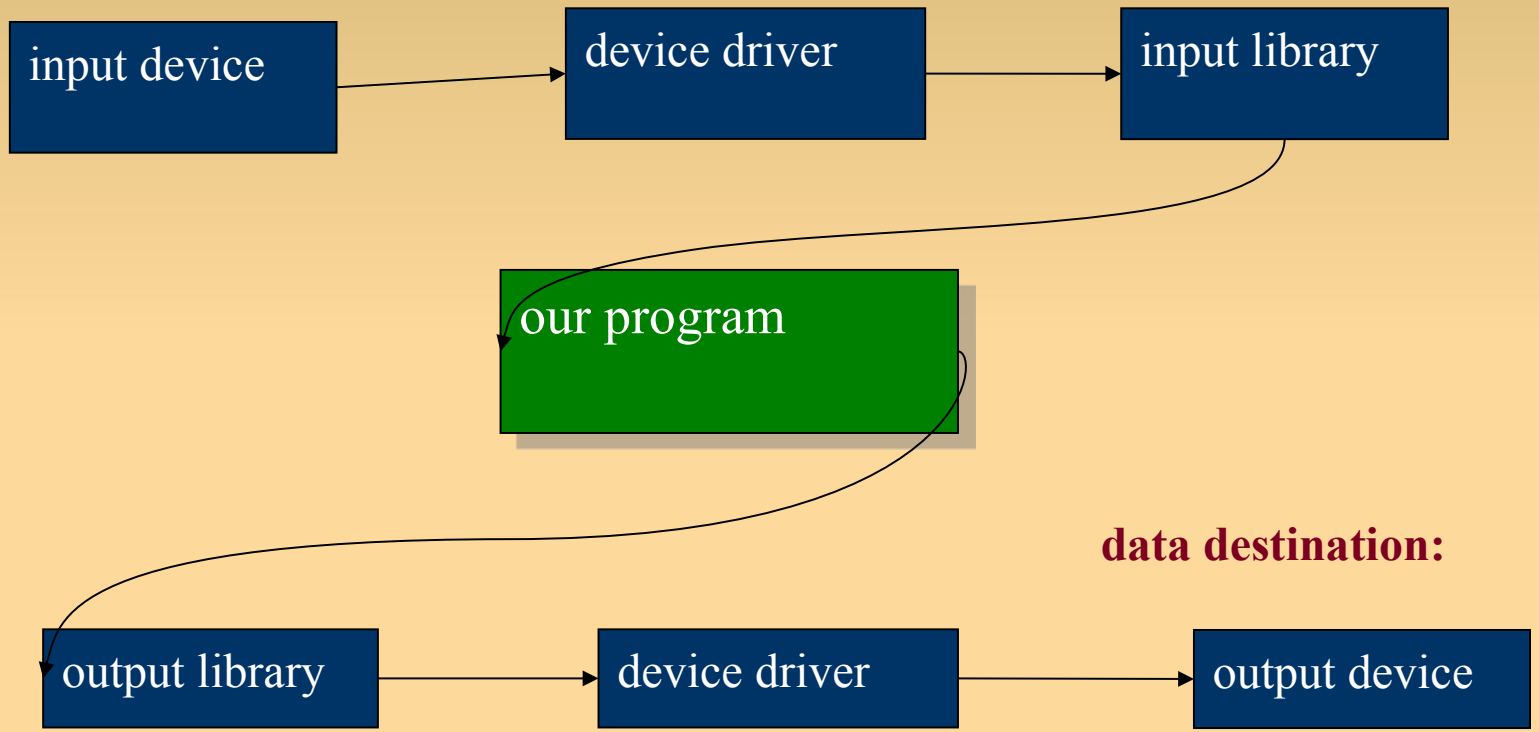
Concept of streams.
I/O errors.
Overloading of streaming operators.

Overview

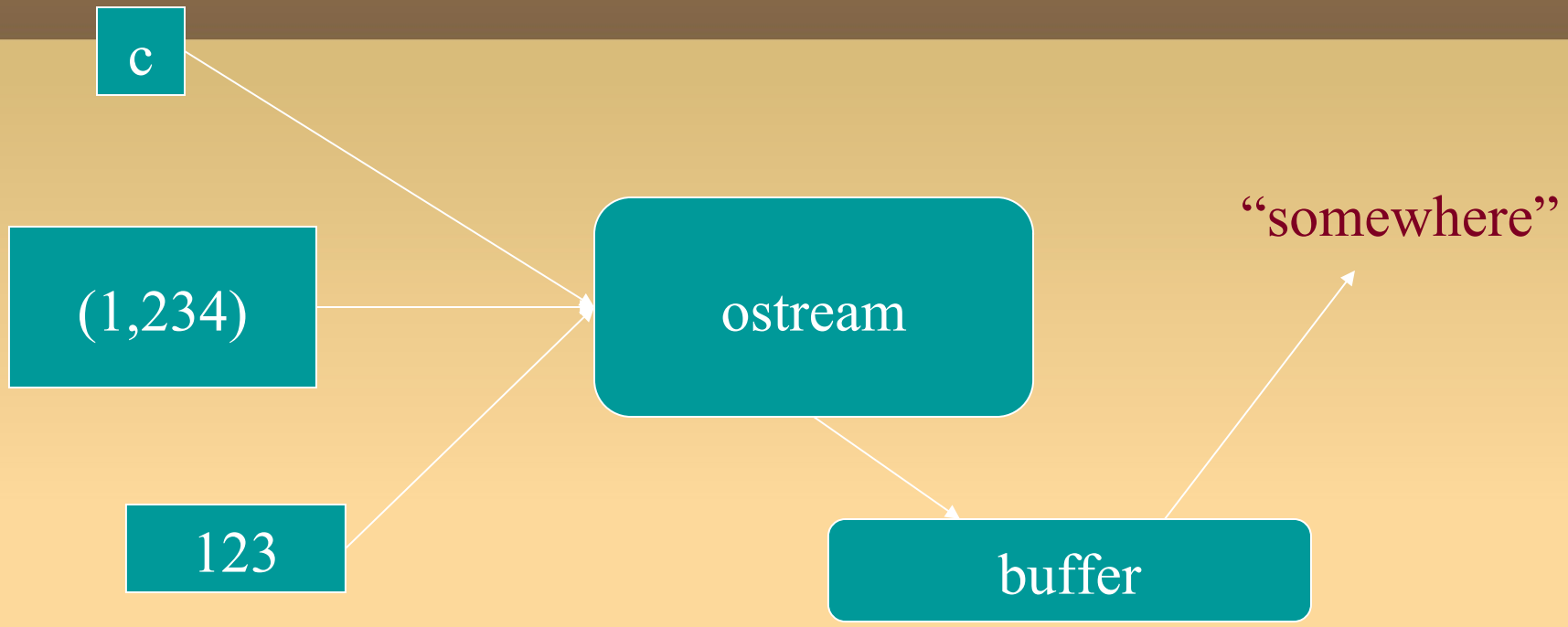
- Fundamental I/O concepts
- Files
  - Opening
  - Reading and writing streams
- I/O errors
- Reading a single integer (example of good programming style)
- User defined output **<<** and input **>>** operators.

Input and Output

data source:

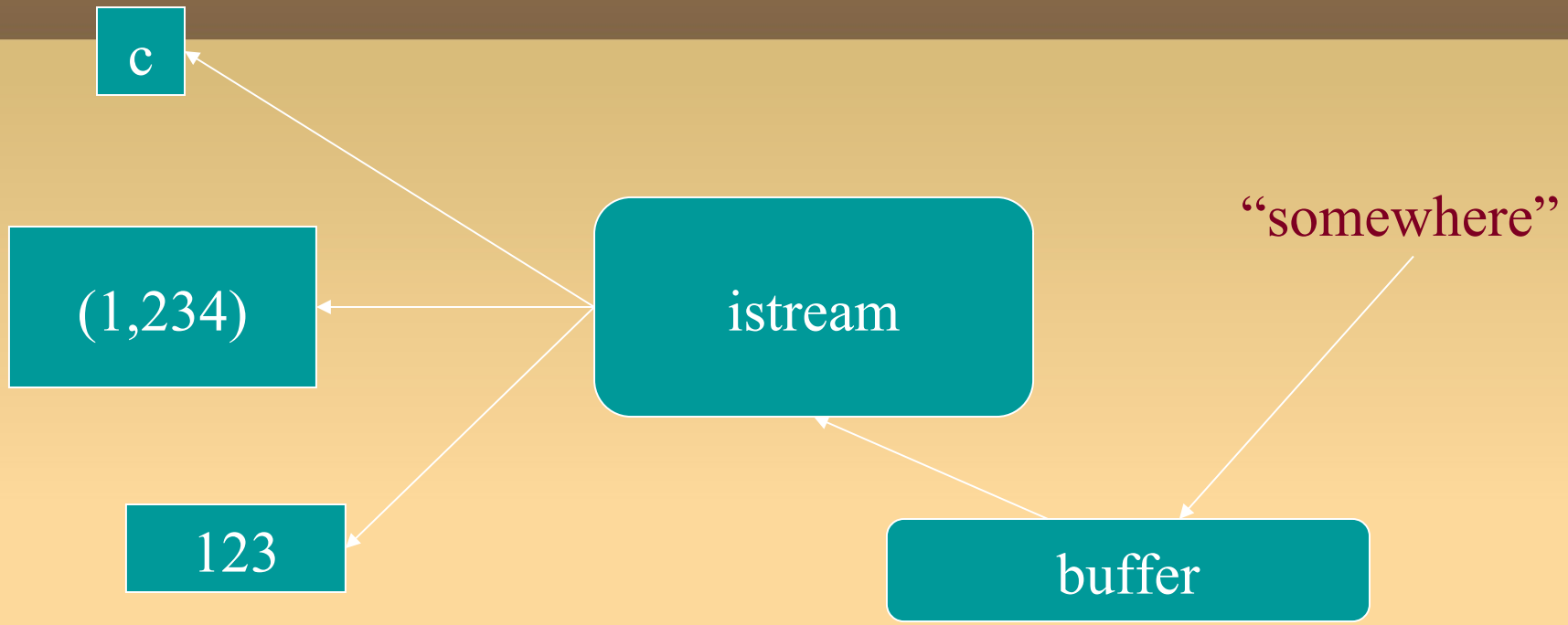


The stream model



- An **ostream** *(output stream)*
  - turns values of various types into byte sequences
  - sends those bytes somewhere
    - *E.g.*, screen, file, main memory, another computer, ip socket

The stream model



- An **istream** *(input stream)*
  - turns byte sequences into values of various types
  - gets those bytes from somewhere
    - *E.g.*, keyboard, file, main memory, another computer, ip socket

The stream model

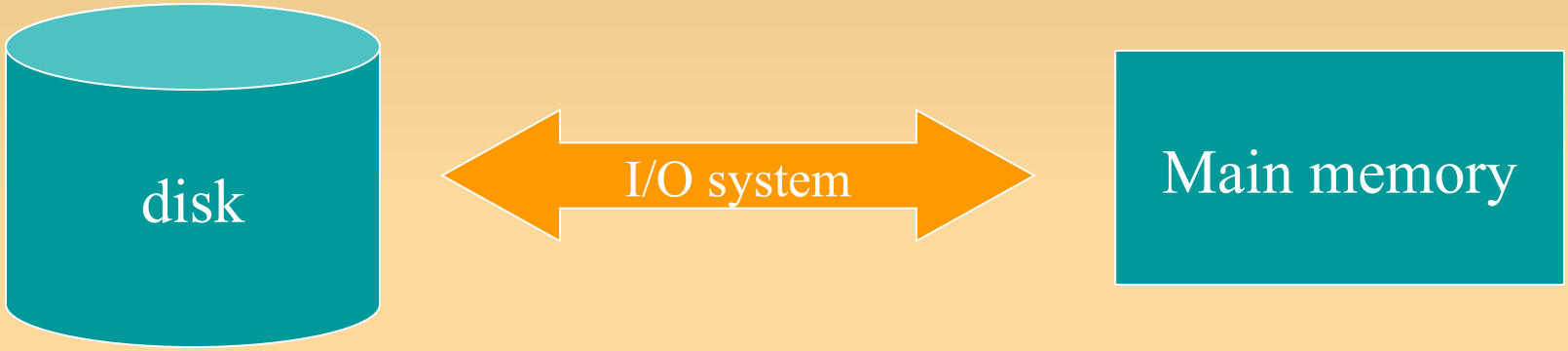
- Reading and writing of typed entities
  - Streaming operators: << (output) and >> (input)
  - Type safe
  - Formatted
  - Extensible
    - You can define your own I/O operations for your own types (by overloading of “<<” and “>>” operators)
  - A stream can be attached to any I/O or storage device

Files

- We turn our computers on and off
  - The contents of memory are “transient data”
- We like to keep our data
  - So we store what we want to preserve on disks or similar permanent storage. Data on permanent storage are “persistent data”.
- A file is a sequence of bytes stored (usually) on permanent storage
  - A file has a name
  - The data on a file has a format (i.e. internal organization, structure)
- We can read / write a file *only if* we know its name and format

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 have to close it
- To write a file
  - We must name it
  - We must open it for writing (this create or re-create file)
  - Then we can write it
  - We *must* close it

Opening a file for reading

```
#include <fstream>  // header of I/O library

int main()
{
    cout << "Please enter input file name: ";
    string name;
    cin >> name;
    ifstream ist (name.c_str());      // ifstream is an "input stream from a file"
                                      // c_str() converts a string to a low-level
                                      // C-style characters' array
    // file of that name is opened for reading

     if ( !ist )  error ("can't open input file " + name);
     // stream ist == false in the case of problem
```

Here **error** *- some method to report an error (e.g. throw exception)*

Opening a file for writing

```
#include <fstream>  // header of I/O library

int main()
{
    cout << "Please enter output file name: ";
    string name;
    cin >> name;
    ofstream ost(name.c_str());     // ofstream is an "output stream to a file"
                                    // c_str() converts string to a low-level
                                    // C-style characters' array
    // file of that name is opened for writing

     if ( !ost ) error ("can't open output file " + name);
     // stream ost == false in the case of problem
```

Remember

- Attempts to read to a file or write from a file – causes errors
- We read in from an input stream (ist >>)
- We write out to an output stream (ost <<)

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       59.2 …
```

- The hours are numbered **0...23**
- No further format is assumed
- Termination
  - Reaching the end of file terminates the reading
  - Anything unexpected in the file terminates the reading *(e.g.* character **q)**

Reading a file

```
struct Reading {      // declare a class to holds the data
        int hour;                   // hour after midnight [0:23]
        double temperature;  // a temperature reading
        Reading(int h, double t) :hour(h), temperature(t) { } // constructor
};

// create a vector of  objects Reading to store the data
vector<Reading> temps;
int hour;  double temperature;
while (ist >> hour >> temperature) {    // read in loop until EOF
        if (hour < 0 || 23 <hour) error  ("hour out of range");        // check
        temps.push_back( Reading(hour,temperature) );    // create, 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

  (could be checked using member functions of I/O stream classe):
  - **good()** *// the operation succeeded*
  - **eof()** *// we hit the end of input ("end of file")*
  - **fail()** *// something unexpected happened*
  - **bad()** *// something unexpected and serious happened*

Example of integers reading "failure"

- "terminator character"
  - 1 2 3 4 5 *
  - State is **fail()**
- "format error"
  - 1 2 3 4 5.6
  - State is **fail()**
- "end of file"
  - 1 2 3 4 5 end of file reached (EOF)
  - 1 2 3 4 5 control-Z (Windows)
  - 1 2 3 4 5 control-D (Unix, MacOS)
  - State is **eof()**
- something really bad
  - e.g. disk error
  - State is **bad()**

I/O error handling

```
// read integers from ist to v until eof() or some terminator term (e.g. 'q')
void fill_vector(istream& ist, vector<int>& v, char term)
{
    int i = 0;
    while (ist >> i) v.push_back(i); // read and store in v until "some failure"
    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 failed: maybe it was terminator character?
    if (ist.fail()) {
      ist.clear();  // clear stream state, so that we can get last element
      char c;   ist>>c;  // read a character, hopefully it was terminator
      if (c != term) {     // unexpected character. Not a terminator
          ist.unget();                          // put that character back
          ist.clear(ios_base::failbit);      // set the state back to fail()
      }
    }
}
```

Throw an exception for bad()

*How to make **ist** to throw an exception if it goes **bad?***

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

"**|**" *is bitwise inclusive OR operator. So, it 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"*

Using that, we can simplify our input loops by no longer checking for **bad()** *state but by catching an exception*

Here **ios_base** *is a library class with many system-dependent static public data-members :*

failbit – *bit telling that stream failed*

badbit – *bit telling that stream is in bad state*

“Reading a single value” program
(About programming style)

```
// first simple and flawed attempt

cout<< "Please enter an integer in the range 1 to 10 ”
      <<“(inclusive): "<<endl;
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"<<endl;
}
```

- Three kinds of problems are possible
  - the user types an out-of-range value
  - getting no value (end of file)
  - user types something of wrong type (i.e., not an integer)

“Reading a single value” program

- What do we want to do in those three cases?
  - handle the problem in the code which reads?
  - 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:

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):"<<endl;
int n = 0;
while (n==0) {
    cin >> n;
    if (cin) {      // we got an integer; now check it:
        if (1<=n && n<=10) break; // OK. Jump out of the loop
        cout << "Sorry, " << n
            << " is not in the [1:10] range; please try again"<<endl;
        n=0;
    } else if (cin.fail()) {        // we found something that wasn't an integer
        cin.clear();
        cout << "Sorry, that was not a number; please try again"<<endl;
        char ch;
        while (cin>>ch && ! isdigit(ch)) ;   // skip 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 {
        error("no input");  // eof or bad: give up
    }
}
// if we reach this line, n is in the range [1:10]
```

Handle everything:

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):"<<endl;
int n = 0;
while (n==0) {
    cin >> n;
    if (cin) {      // we got an integer; now check it:
        if (1<=n && n<=10) break; // OK. Jump out of the loop
        cout << "Sorry, " << n
            << " is not in the [1:10] range; please try again"<<endl;
        n=0;
    } else if (cin.fail()) {        // we found something that wasn't an integer
        cin.clear();
        cout << "Sorry, that was not a number; please try again"<<endl;
        char ch;
        while (cin>>ch && ! isdigit(ch)) ;   // skip 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 {
        error("no input");   // eof or bad: give up
    }
}
// if we reach this line, n is in the range [1:10]
```

What's a mess!

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 “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***
  - **void skip_to_int()**; *// we found some "garbage" character*
    *// so skip until we find an **int***

Separate functions that do the logically separate actions.

Call hierarchy: **get_int(int, int)** => **get_int()** => **skip_to_int()**

Start programming from back to beginning in call hierarchy

Skip “garbage”

```
void skip_to_int()
{
    if (cin.fail()) {          // we found something that wasn't an integer
        cin.clear();           // reset stream state to good()
        char ch;
        while (cin>>ch) {      // read char in loop and skip non-digits
            if (isdigit(ch)) { // if in ch there is a digit 0...9
                cin.unget();   // put the digit back,
                               // so that we can read the number later
                return;        // exit
            }
        }
    }
    error("no input");         // eof or bad: give up
}
```

Get (any) integer

```
int get_int()
{
    int n = 0;
    while (true) {      // infinite loop
      if (cin >> n) return n;  // OK. Integer was read it. Return it. Exit.
      cout << "Sorry, that was not an integer; please try
    again"<<endl;
      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): "<<endl;
    while (true) {
      int n = get_int();
      if (low<=n && n<=high) return n;  // OK. In the range
      cout<< "Sorry, "
          << n << " is not in the [" << low << ':' << high
          << "] range; please try again: "<<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 will improve

// let's pass as arguments "range" and "dialog" (parametrization)

```
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"<<endl;
```

Parameterize
(pass by parameters)

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

- incomplete parameterization as **get_int()** still “talking”
- “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 streamers

- In C++ build-in operators could be redefined (overloaded) for user defined types (classes)
- So, streaming operators “<<” and “>>” also could be overloaded

User-defined output: operator << ()

- Could be implemented only as non-member function

  as it has library type “stream” as left operand and this class couldn't be modified:

```
friend ostream& operator << (ostream& os, const Date& d) {
    return os << '(' << d.y << ',' << d.m<< ',' << d.d << ')';
}
```

Here **friend** means that despite the function **operator <<** is not a member function of our class **Date** it has an access to this class private members

Note: return type is reference. To be able to organize streaming “chains”

Use

```
void do_some_printing (Date d1, Date d2)
{
  cout << d1 << endl;  // means call of function operator << (cout, d1) ;
  cout << d1 << d2 << endl; // means (cout << d1)  <<  d2
}
```

User-defined input: operator >>()

```
// Read date in format: ( year , month , day ) E.g. (2021,03,08)

friend istream& operator >> (istream& is, Date& dd)
{
    int y, d, m;
    char c1, c2, c3, c4;
    is >> c1 >> y >> c2 >> m >> c3 >> d >> c4;
    if (!is) return is;    // there was some type mismatch, so just leave
    if ( c1 != '(' || c2 != ',' || c3 != ',' || c4 != ')' ) {   // oops: format error
      is.clear(ios_base::failbit);   // format was wrong: set state to fail()
      return is;                     // and leave
    }
    dd = Date(y, Month(m), d);   // OK. Update dd
    return is;                   // and leave with is in the good() state
}
```

Next talk

- vector
- Pointers
- Free store