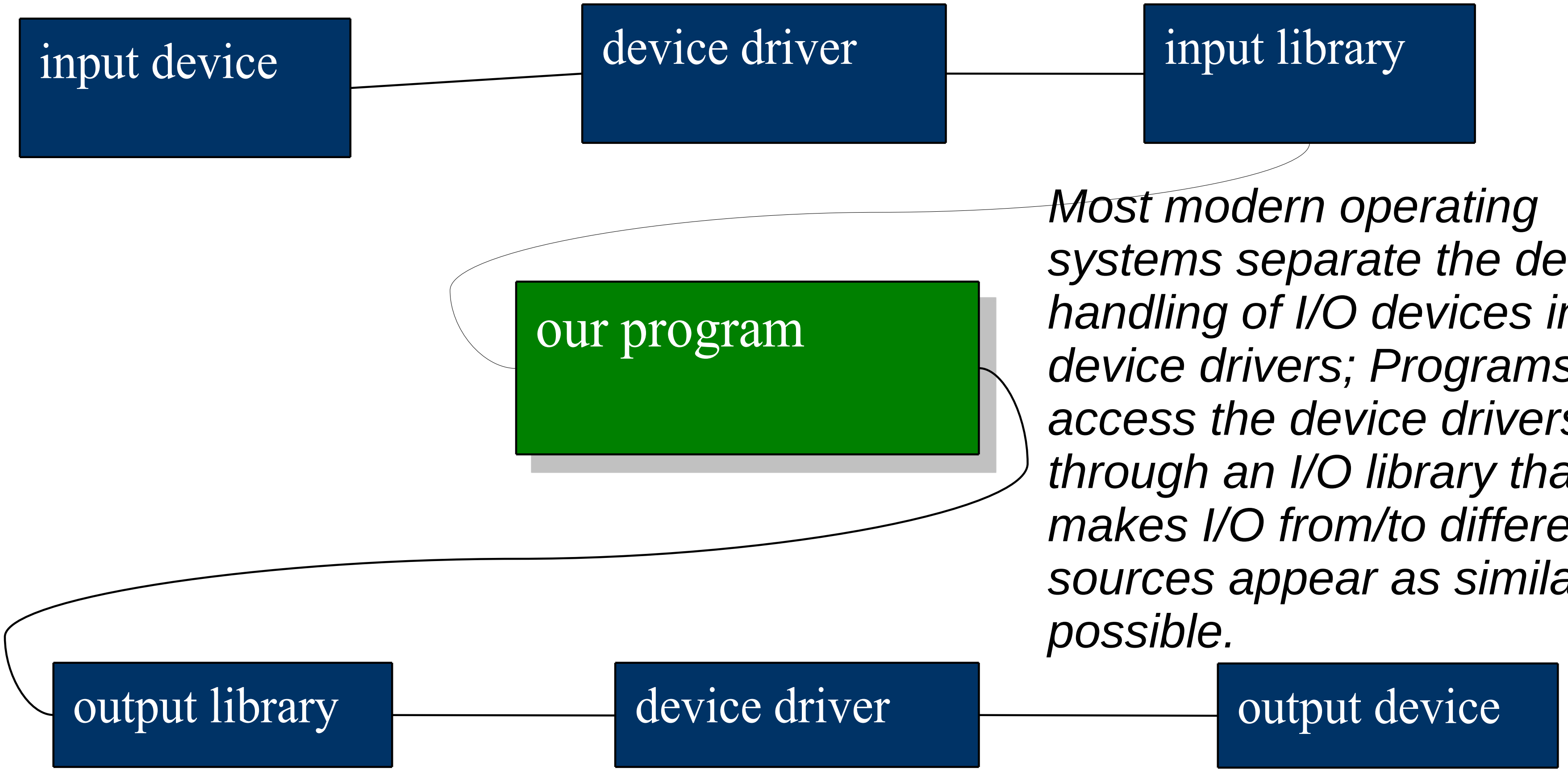
Chapter 10: Input and Output Streams

Plan for today

- We will talk about:
  - The I/O stream model
  - Files:
    - Opening a file
    - Reading and writing a file
  - I/O error handling
  -

Input and Output

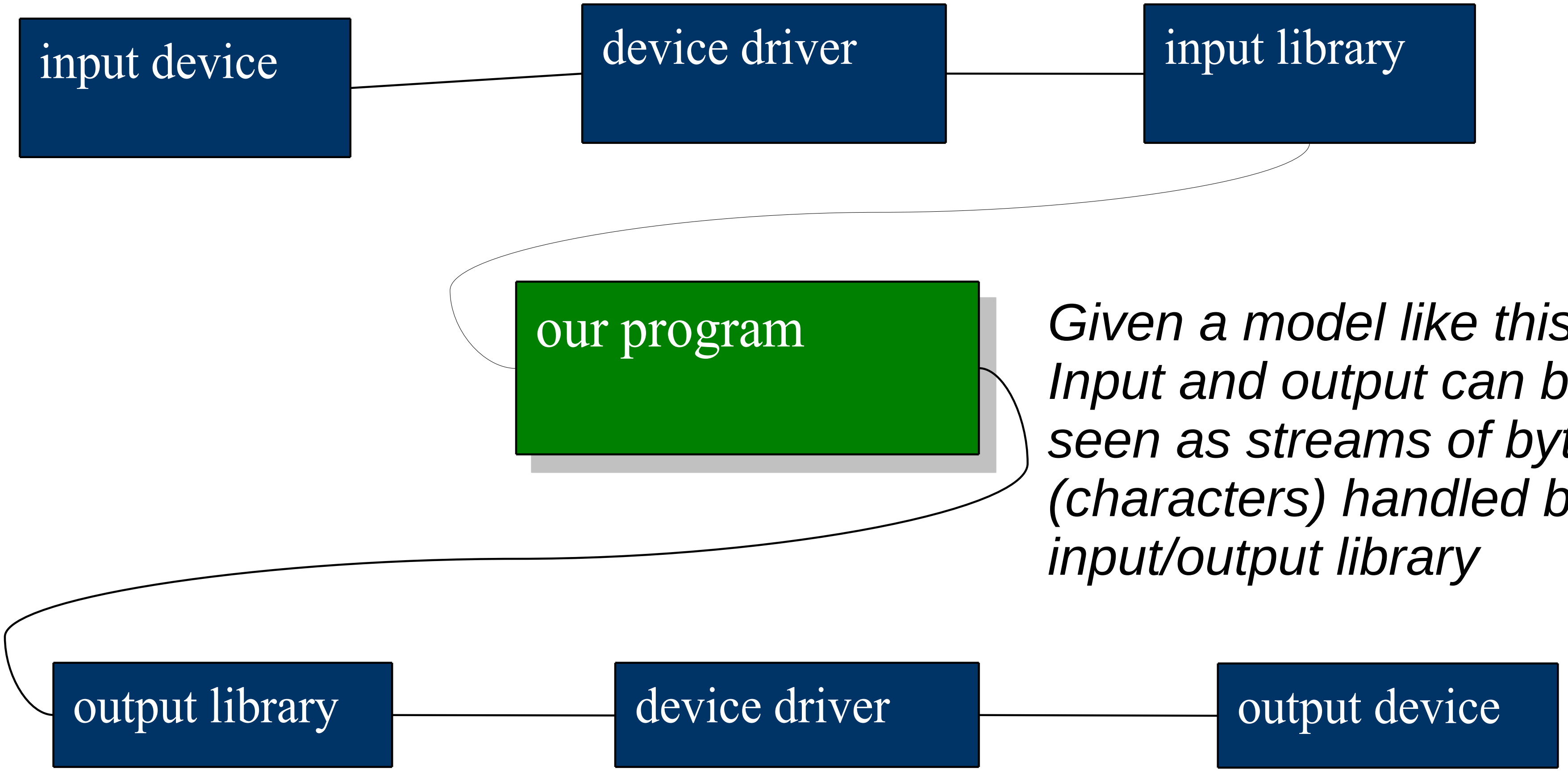
data source:



Most modern operating systems separate the detailed handling of I/O devices into device drivers; Programs then access the device drivers through an I/O library that makes I/O from/to different sources appear as similar as possible.

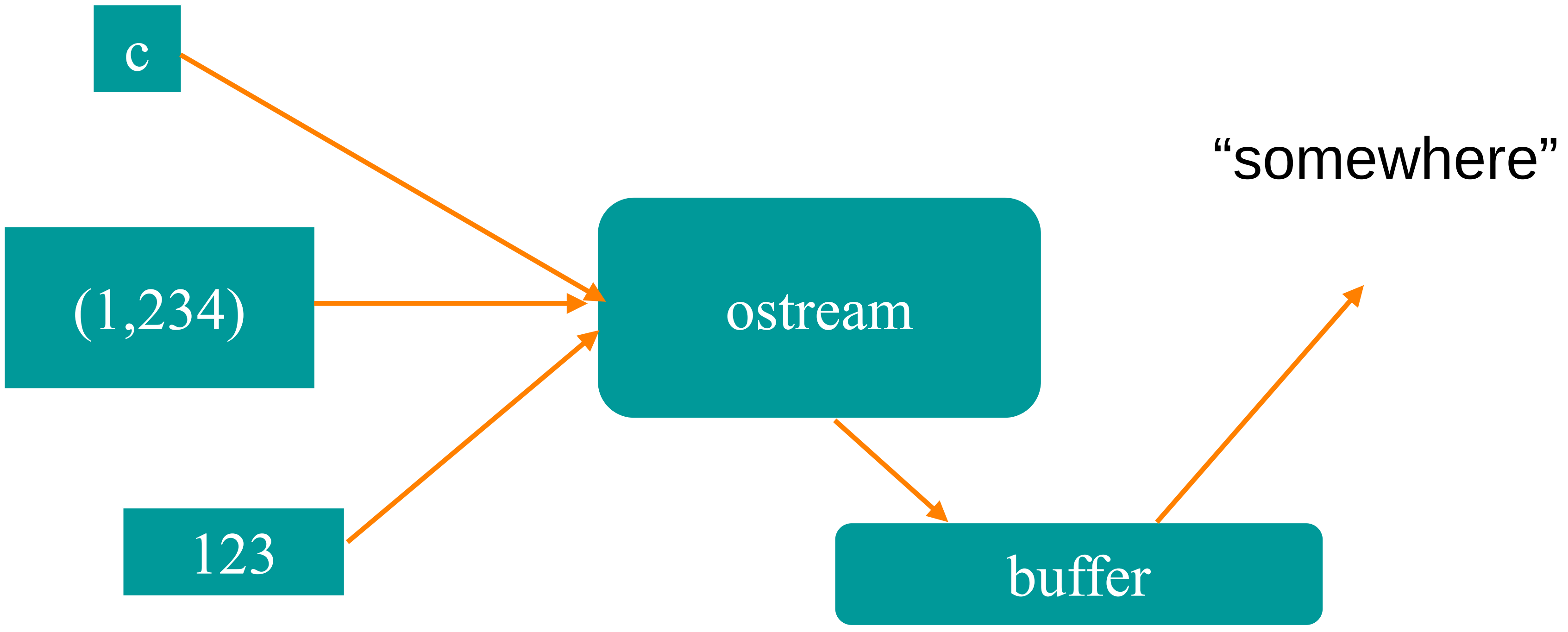
Input and Output

data source:



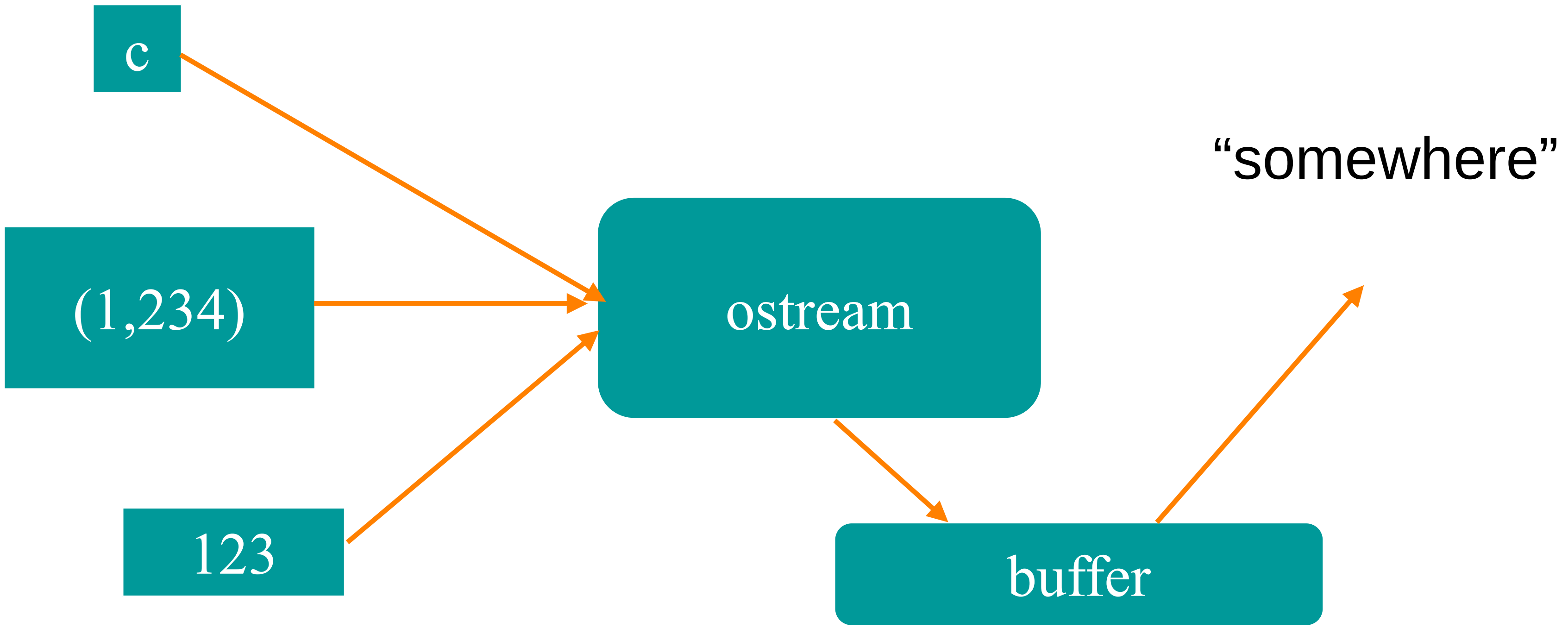
Given a model like this: Input and output can be seen as streams of bytes (characters) handled by the input/output library

The output stream model



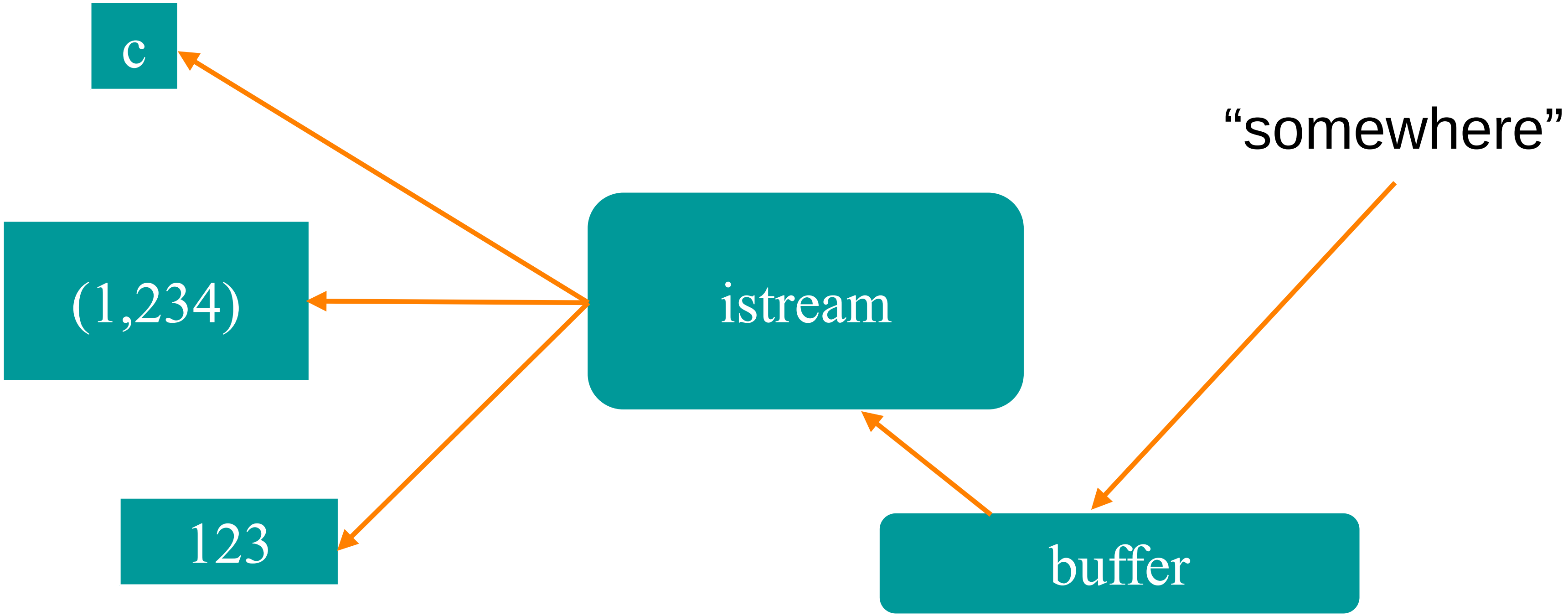
An **ostream**:
- turns values of various types into character sequences
- sends those characters "somewhere" (console, file, main memory, another computer, etc.)

The output stream model



`Buffer` is a data structure that the `ostream` uses internally to store the data we give it while communicating with the operating system.
It is important for performance. Sometimes we may notice a delay between our writing to an ostream and the characters appearing at their destination.

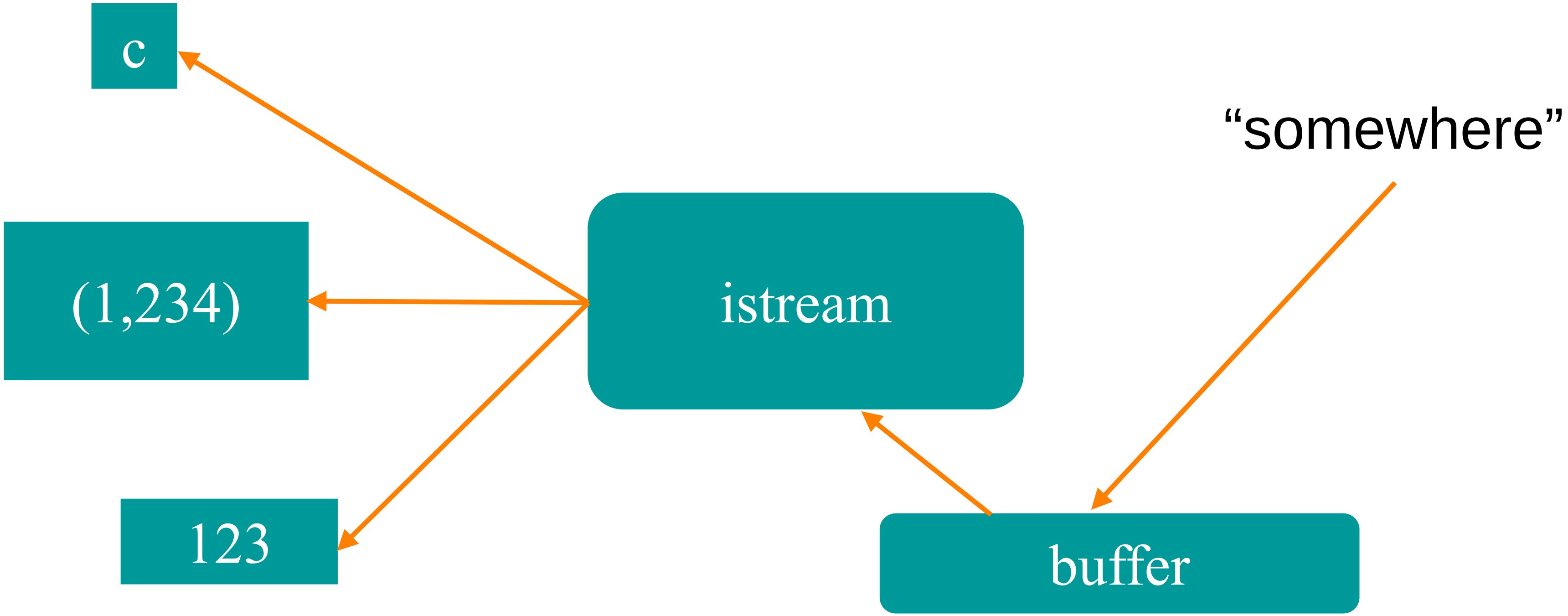
The input stream model



An **`istream`**:

- turns character sequences into values of various types
- gets those characters from "somewhere" (console, file, main memory, another computer, etc.)

The input stream model



With an `istream`, the buffering can be quite visible.
Example: when the user types on a keyboard, until they press Enter, they can modify the entered text.

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

- 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

At the fundamental level, a file is a sequence of bytes numbered from 0 upwards.

Files

- General model

Files
(sequences of bytes)

iostreams

Objects
(of various types)

For a file:

- An `ostream` converts objects in main memory into streams of bytes and writes them to disk
- An `istream` does the opposite: it takes a stream of bytes from disk and composes objects from them

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 (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 (typically done implicitly)

Opening a file for reading

```
// …
int main()
{
    cout << "Please enter input file name: ";
    string iname;
    cin >> iname;
    ifstream ist {iname};   // 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};   // 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);
// …
}
```

Example

- Assume we have a file that contains a sequence of pairs representin hours and temperature readings
- The hours are numbered 0 … 23
- No further format is assumed
- Termination is upon reaching the end of the file, or anything unexpected is read.

0 60.7

1 60.6

2 60.3

3 59.22

see program temperatureReadings.cpp

In-class practice (exercise 9)

- Write a program that takes two files containing sorted whitespace-separated words and merges them into one file, preserving the sorted order.

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; 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()
} } }
```

Sequence of integers: 4 2 9 8 1 7 *

Throw an exception for bad()

```
// 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

Sequence of integers: 4 2 9 8 1 7 *

Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html

- Class textbook