deep se
dependable evolvable pervasive software engineering group

Streams & I/O

15/11/2024

Federica Filippini, Luca Pozzoni,
Giovanni Quattrocchi, Danilo Ardagna

Politecnico di Milano
{nome.cognome}@polimi.it

POLITECNICO
DI MILANO

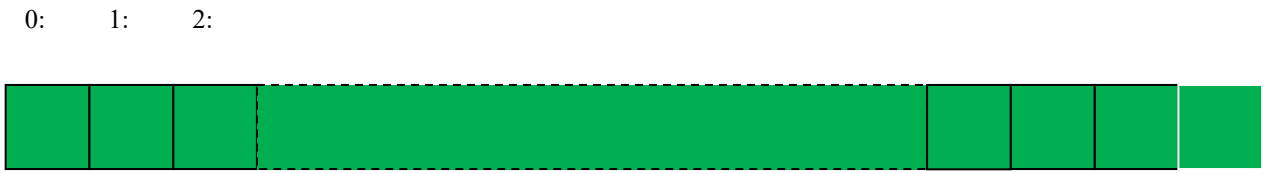
Content

- Fundamental I/O concepts
- Files
  - Opening
  - Reading and writing streams
- File modes and Binary I/O
- String streams
  - Line-oriented input
- Examples

Files

- We turn our computers on and off
  - The contents of its main memory is transient
- Data needs to be preserved
  - It must be stored on disks and similar permanent storage devices
- 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

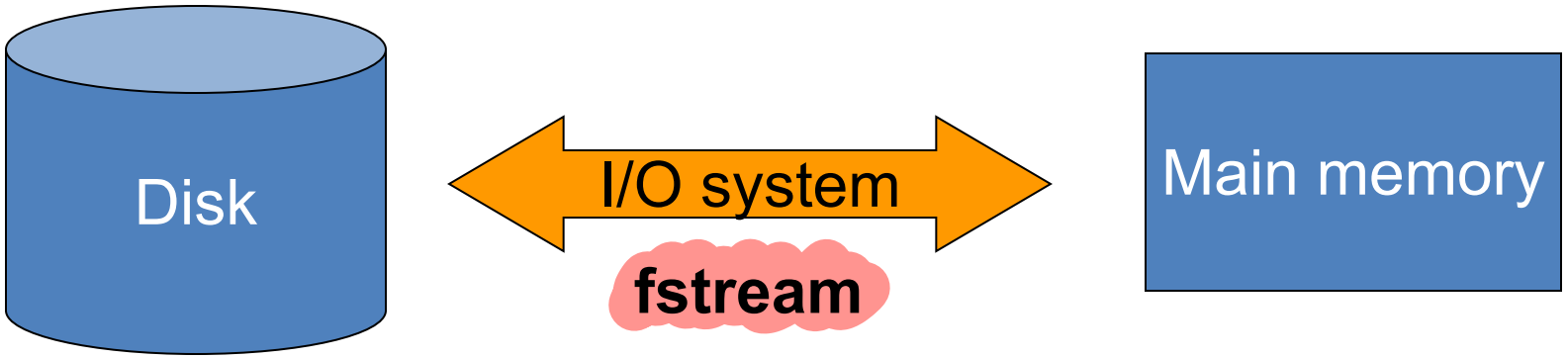
A file



- 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 corresponding to "123.45" might be interpreted as the floating-point number 123.45

Files

- General model



Files
(sequences of bytes)

Objects
(of various types)

Files

- To **read** a file
  - We must know its name
  - We must open it (for reading)
  - Then we can read it
  - Once finished, we must close it
    - That is typically done implicitly (when the stream object is destroyed)
- 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
  - Once finished, we must close it
    - That is typically done implicitly (when the stream object is destroyed)

Opening a file for reading

```
// ...
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);
    // ...
}
```

Implicit close

- When an **fstream** object goes out of scope, the file it is bound to is automatically closed

```
if (read) {
        // create input and open the file
        ifstream input{name};
        if (input) {  // if the file is ok, "process" this file
                process(input);
        } else
                cerr << "couldn't open: " + ;
        } // input goes out of scope and is destroyed on each iteration
}
```

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 from **0** to **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., character ‘q’

Reading a file

Syntax.

```
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;
ifstream ist{fname};
while (ist >> hour >> temperature) {    // read
        if (hour < 0 || 23 <hour)
                cout << "hour out of range" <<endl;
        temps.push_back( Reading{hour, temperature} );  // store
}
```

looks like we would do with cin to read from the command line.

```
ifstream ist{fname};
while (ist >> hour >> temperature) {.   // read
       temps.push_back( Reading{hour, temperature} ); // store
}
```

```
0  60.7 \n
q  60.6
2  60.3
3  59.22
```

```
0 | 60.7 \n
q  60.6
2  60.3
3  59.22
```

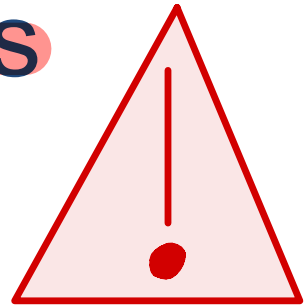
```
0  60.7 \n
q  60.6
2  60.3
3  59.22
```

```
0  60.7
q  60.6
2  60.3
3  59.22
```

The stream will stop reading, because of q.

We'll see smthg more complex, because this is very basic.

No Copy or Assign for I/O Objects



- **We cannot copy or assign objects of the IO types:**

```
ofstream out1, out2;
out1 = out2; // error: cannot assign stream objects ofstream
ofstream print(ofstream); // error: can't initialize the ofstream param
out2 = print(out2); // error: cannot copy stream objects
```

Because we can't copy the IO types, we cannot have a parameter or return type that is one of the stream types

- Functions that do IO typically pass and return the stream through **references**
- Reading or writing an IO object changes its state, so the reference **must not be const**

because it changes the state of the stream.

```
void print(ofstream&); // OK
```

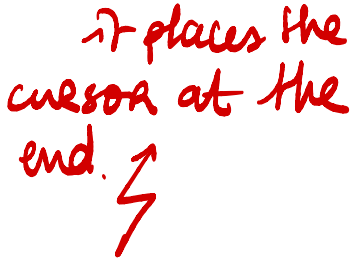
Use

```
void do_some_printing(Date d1, Date d2)
{
  cout << d1; // means:
              // operator<<(cout,d1) ;

  cout << d1 << d2;
              // means:
              // (cout << d1)  <<  d2; same as:
              // (operator<<(cout,d1)) <<  d2; same as:
              //  operator<<((operator<<(cout,d1)), d2);
}
```

File Modes and Binary I/O

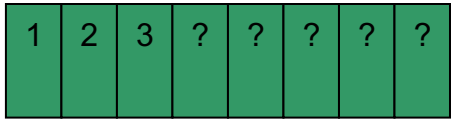
File open modes

- By default, an ifstream opens its file for reading
- By default, an ofstream opens its file for writing.
- Alternatives:
  - ios_base::**app** // append (i.e., output adds to the end of the file)
  - ios_base::**ate** // "at end" (open and seek to end)
  - ios_base::**binary** // binary mode – beware of system specific behavior
  - ios_base::**in** // for reading
  - ios_base::**out** // for writing
  - ios_base::**trunc** // truncate file to 0-length
- A file mode is optionally specified after the name of the file:
  - **ofstream** of1 {name1}; // defaults to **ios_base::out**
  - **ifstream** if1 {name2}; // defaults to **ios_base::in**
  - ofstream ofs {name, **ios_base**::**app**}; // append rather than overwrite
  - fstream fs {"myfile", **ios_base::in | ios_base::out**}; // both in and out

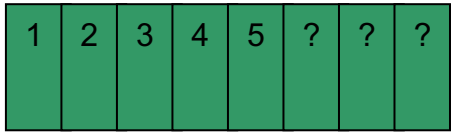


Text vs. binary files

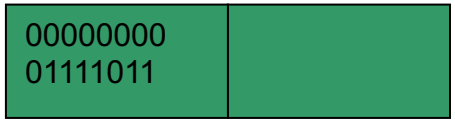
123 as characters:



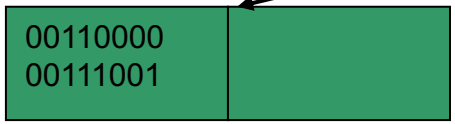
12345 as characters:



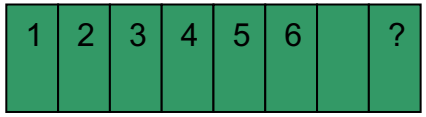
123 as binary:



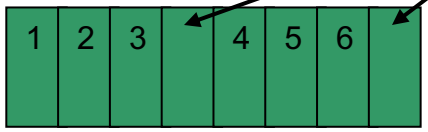
12345 as binary:



123456 as characters:



123 456 as characters:



In binary files, we use sizes to delimit values

In text files, we use separation/termination characters

Text vs. binary

- Use text when you can
  - You can read it (without a fancy program)
  - You can debug your programs more easily
  - Text is portable across different systems
  - Most information can be represented reasonably as text
- Use binary when you must
  - E.g. image files, sound files

except for example image, or sound files.

String Streams

String streams

- A **stringstream** reads/writes from/to a **string** rather than a file or a keyboard/screen

```
double str_to_double(string s)
// if possible, convert characters in s to floating-point value
{
        istringstream is {s};// make a stream so that
                             // we can read from s
        double d;
        is >> d;
        if (!is) error("double format error: ",s);
        return d;
}

double d1 = str_to_double("12.4");          // testing
double d2 = str_to_double("1.34e-3");
double d3 = str_to_double("twelve point three");// error
```

Reads up to the first space.

performs the conversion : like when we type in the command line.

String streams

- See textbook for **ostringstream**

- String streams are very useful for
  - formatting into a fixed-sized space (think GUI)
  - for extracting typed objects out of a string

Type vs. line

- Read a string

```
string name;
cin >> name;
cout << name << '\n';
```

// input: **Dennis Ritchie**
// output: **Dennis**

That's a limitation of cin.

- Read a line

```
string name;
getline(cin,name);
cout << name << '\n';
// now what?
// maybe:
istringstream ss{name};
ss >> first_name;
ss >> second_name;
```

// input: **Dennis Ritchie**
// output: **Dennis Ritchie**

1st use "getline" to read all the line. 2nd separate elements from the line.

It reads from cin, save in the second parameter (name), up to the end of the line.

parses the line: it separates the elements. Visually that's what we do.

Examples

Example 1: reading a CSV file

- An istringstream is often used when we have some work to do on an entire line, and other work to do with individual words within a line

CSV

```
Morgan,2015552368,8625550123
Drew,9735550130
Lee,6095550132,2015550175,8005550000
```

We cannot write something like

ifs >> name >> n1 >> n2;

because there is NO space!

```
// members are public by default
struct PersonInfo {
   string name;
   vector<string> phones;
};
```

```
vector<PersonInfo> people; // will hold all the records from the input
string line;
ifstream data("data.csv");

// read the input a line at a time until cin hits end-of-file (or another error)
while ( getline(data, line) ) {
    PersonInfo info; // create an object to hold this record's data
    istringstream record(line); // bind record to the line we just read

    // read the name
    // note that we are changing the delimenter of getline to ","
    getline(record, info.name, ',') ;
    string phone;
    // read the phone numbers
    while (getline(record, phone, ',')){
      info.phones.push_back(phone); // and store them
    }
    people.push_back(info); // append this record to people
}
```

line is: "Noe, 12345,6789"

we put the single line in line.

Idea: 1) loop over the lines
2) loop over the elts in the line

we put here

we use get line again but with "," delimiter.

→ of previous slide, we might have 2 phone numbers.

Example 2: A Word Transformation Map

Write a program that given one string, transforms it into another. The input to our program is two files. The first file contains **rules** that we will use **to transform** the **text** in the second file. Each rule consists of a word that might be in the input file and a phrase to use in its place.

word-transformation file:

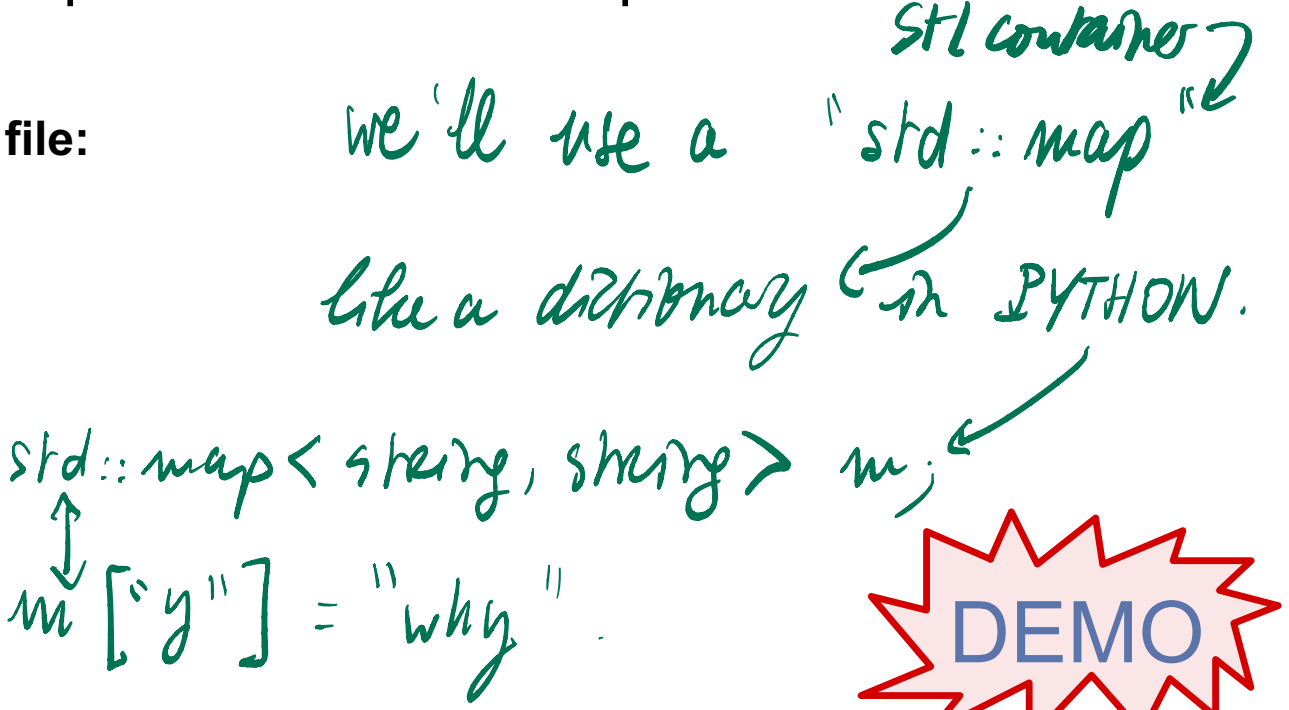
```
y why
r are
u you
```

second file:

```
where r u
```

output file:

```
where are you
```



An example: A Word Transformation Map

```
void word_transform(ifstream &map_file, ifstream &input)
{
auto trans_map = buildMap(map_file);
string text;
while (getline(input, text)) {
        istringstream stream(text);
        string word;
        bool firstword = true;
        while (stream >> word) {
                if (firstword)
                        firstword = false;
                else
                        cout << " ";
                cout << transform(word, trans_map);
        }
        cout << endl;
}
}
```

An example: A Word Transformation Map

```
map<string, string> buildMap(ifstream &map_file)
{
map<string, string> trans_map;
string key;
string value;
while (map_file >> key && getline(map_file,
                                  value))
      if (value.size() > 1)
            trans_map[key] = value.substr(1);
      else
            cout << "no rule for " + key << endl;
return trans_map;
}
```

An example: A Word Transformation Map

```
const string &
transform(const string &s, const map<string,
                                     string> &m)
{
auto map_it = m.find(s);
if (map_it != m.cend())
      // if this word is in the transformation map
      return map_it->second;
else
      return s;
}
```

Readings

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

User-defined input: operator>>()

```
istream& operator>>(istream& is, Date& dd)
  // Read date in format: year month day
{
  int y, d, m;
  if (is >> y >> m >> d){
       dd = Date{y,m,d}; // update dd
  }
  return is;
}
```

Binary files

```
int main()    // use binary input and output
{
 cout << "Please enter input file name\n";
 string iname;
 cin >> iname;
 ifstream ifs {iname, ios_base::binary};// note: binary
 if (!ifs) error("can't open input file ", iname);

 cout << "Please enter output file name\n";
 string oname;
 cin >> oname;
 ofstream ofs {oname, ios_base::binary};// note: binary
 if (!ofs) error("can't open output file ", oname);


 // "binary" tells the stream not to try anything clever operation
 //  with the bytes
```

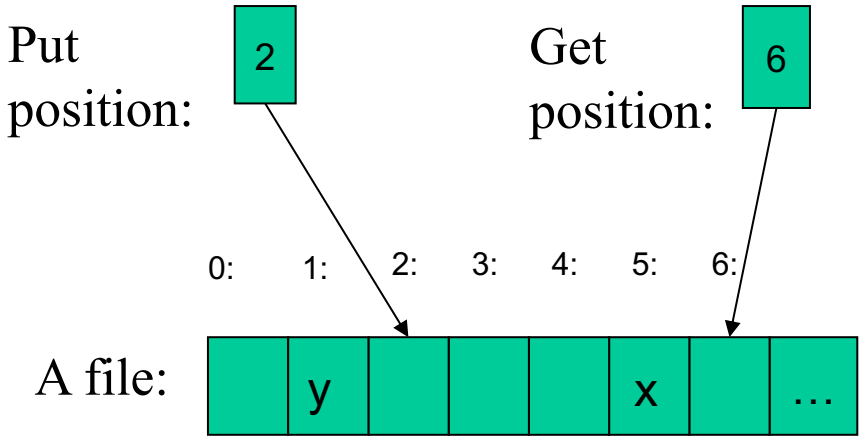
Binary files

```
  vector<int> v;

  // read from binary file:
  for (int i; ifs.read(as_bytes(i), sizeof(int)); )
  // note: reading bytes
        v.push_back(i);

  // … do something with v …

  // write to binary file:
  for(int i=0; i<v.size(); ++i)
        ofs.write(as_bytes(v[i]), sizeof(int));// note: writing
                                                // bytes
  return 0;
}

// For now, treat as_bytes() as a primitive
// Warning!  Beware transferring between different systems
```

Positioning in a filestream

- To support random access, the system maintains a marker that determines where the next read or write will happen
- We also have two functions:
  - One **repositions the marker** by seeking to a given position
  - The second **tells the current position** of the marker
- The library actually defines two pairs of seek and tell functions:
  - One pair is used by input streams, the other by output streams
  - The input and output versions are distinguished by a suffix that is either a **g** (“getting”, i.e. reading data), or **p** (“putting”, i.e. writing data)

Positioning in a filestream




```
fstream fs {name}; // open for input and output
// …
fs.seekg(5);  // move reading position ( 'g' for 'get') to 5 (the 6th character)
char ch;
fs>>ch;       // read the x and increment the reading position to 6
cout << "sixth character is " << ch << '(' << int(ch)
     << ")\n";
fs.seekp(1);  // move writing position ( 'p' for 'put') to 1 (the 2nd character)
fs<<'y';                  // write and increment writing position to 2
```

Positioning in a filestream

- We can use only the **g** versions on an **istream** and on the types that inherit from it, **ifstream** and **istringstream**
- We can use only the **p** versions on an **ostream** and on the types that inherit from it, **ofstream** and **ostringstream**
- An **iostream**, **fstream**, or **stringstream** can both **read** and **write** the associated stream; we can use either the **g** or **p** versions on objects of these types

There Is Only One Marker

- The fact that the library distinguishes between the "**putting**" and "**getting**" versions of the **seek** and **tell** functions can be misleading

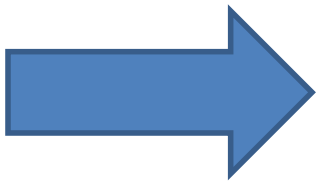
- Even though the library makes this distinction, **it maintains only a single marker in a stream** — there are no distinct **read** and **write** markers

Repositioning the Marker

```
// set the marker to a fixed position
seekg(new_position);  // set the read marker to the given pos_type
                      // location
seekp(new_position);  // set the write marker to the given pos_type
                      // location
// offset some distance ahead of or behind the given starting point
seekg(offset, from);  // set the read marker offset distance from
from seekp(offset, from);  // offset has type off_type
```

Reading and writing to the same file

abcd
efg
hi
j



abcd
efg
hi
j
5 9 12 14

Reading and writing to the same file

```
int main()
{
        // open for input and output and preposition file pointers to end-of-file
        // file mode argument
        fstream inOut("copyOut", fstream::ate | fstream::in |
                                          fstream::out);
        if (!inOut) {
                cerr << "Unable to open file!" << endl;
                return EXIT_FAILURE;//EXIT_FAILURE
        }
        // inOut is opened in ate mode, so it starts out positioned at the end
        auto end_mark = inOut.tellg();// remember original end-of-file
                                          // position
        inOut.seekg(0, fstream::beg);  // reposition to the start of the file
        size_t         cnt = 0;  // accumulator for the byte count string line;
        string line;  // hold each line of input
```

Reading and writing to the same file

```
    // while we haven't hit an error and are still reading the original
    // data and can get another line of input
    while (inOut && inOut.tellg() != end_mark
            && getline(inOut, line))
    {
            cnt += line.size() + 1; // add 1 to account for the newline
            auto mark = inOut.tellg(); // remember the read position
            inOut.seekp(0, fstream::end); // set the write marker to the end
            inOut << cnt; // write the accumulated length
            // print a separator if this is not the last line
            if (mark != end_mark) inOut << " ";
            inOut.seekg(mark); // restore the read position
    }
    inOut.seekp(0, fstream::end); // seek to the end
    inOut << "\n"; // write a newline at end-of- file
    return 0;
}
```

Positioning

- Whenever you can
  - Use simple streaming
    - Streams/streaming is a very powerful metaphor
    - Write most of your code in terms of “plain” istream and ostream
  - Positioning is far more error-prone
    - Handling of the end of file position is system dependent and basically unchecked

Using ostringstreams

- An **ostringstream** is useful when we need to build up our output a **little at a time** but **do not** want to print the output **until later**
- For example, we might want to validate and reformat the phone numbers we read in the previous example
  - If all the numbers are valid, we want to print a new file containing the reformatted numbers
  - If a person has any invalid numbers, we won't put them in the new file. Instead, we'll write an error message containing the person's name and a list of their invalid numbers

Using ostringstreams

```
for (const auto &entry : people) { // for each entry in people
    ostringstream formatted, badNums; // objects created on each loop
    for (const auto &nums : entry.phones) { // for each number
        if (!valid(nums)) {
                badNums << " " << nums; // string in badNums
        } else
        // "writes" to formatted's string
                formatted << " " << format(nums);
        }
        if (badNums.str().empty()) // there were no bad numbers
                os << entry.name << " " // print the name
                << formatted.str() << endl; // and reformatted numbers
        else
                // otherwise, print the name and bad numbers
                cerr << "input error: " << entry.name
                << " invalid number(s) " << badNums.str() << endl;
}
```

References

- Lippman Chapters 8, 17

Credits

- Bjarne Stroustrup. www.stroustrup.com/Programming