Introduction to C++ Part 3
Steams, Input, Formatting

Append to files

- Use ios_base::app

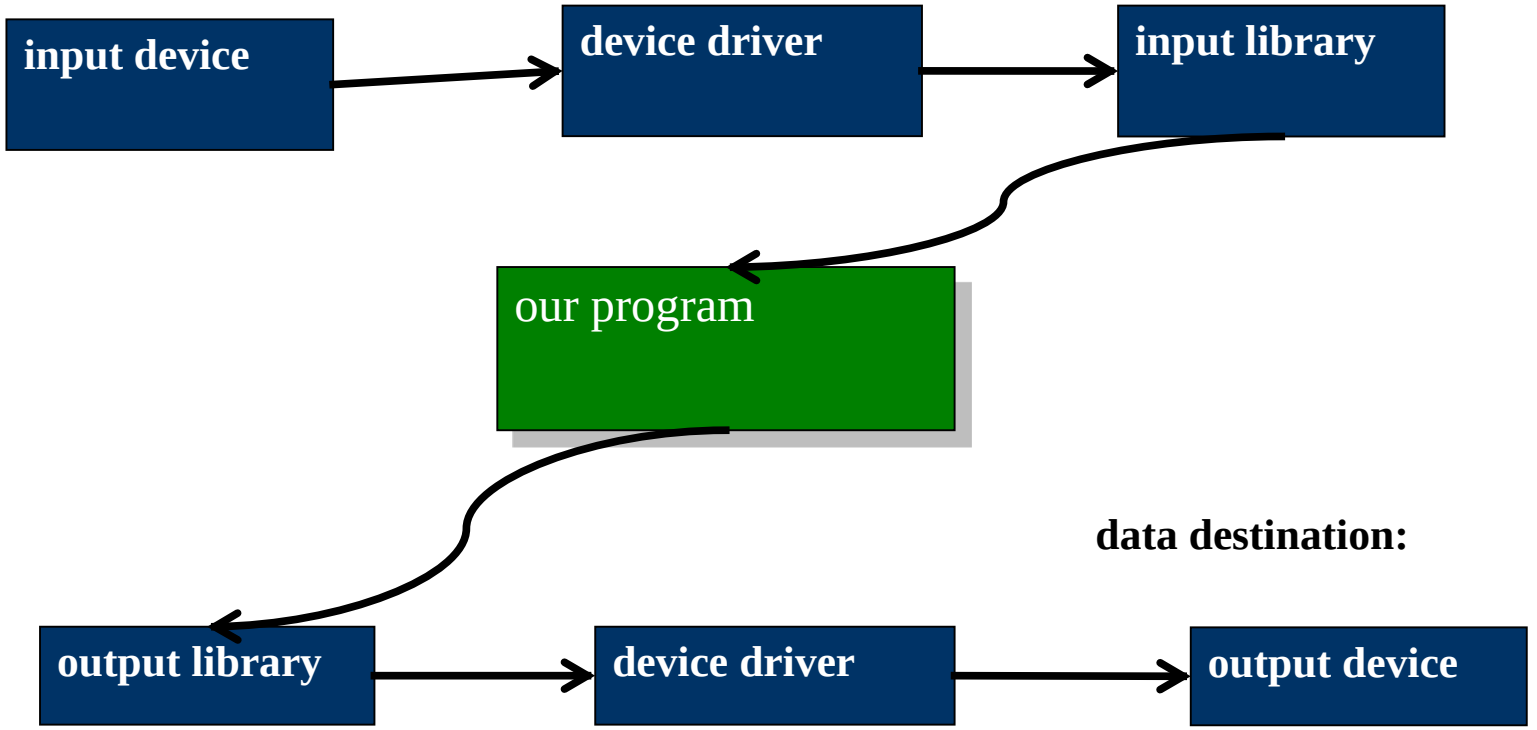
File open modes

- 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*

Streams

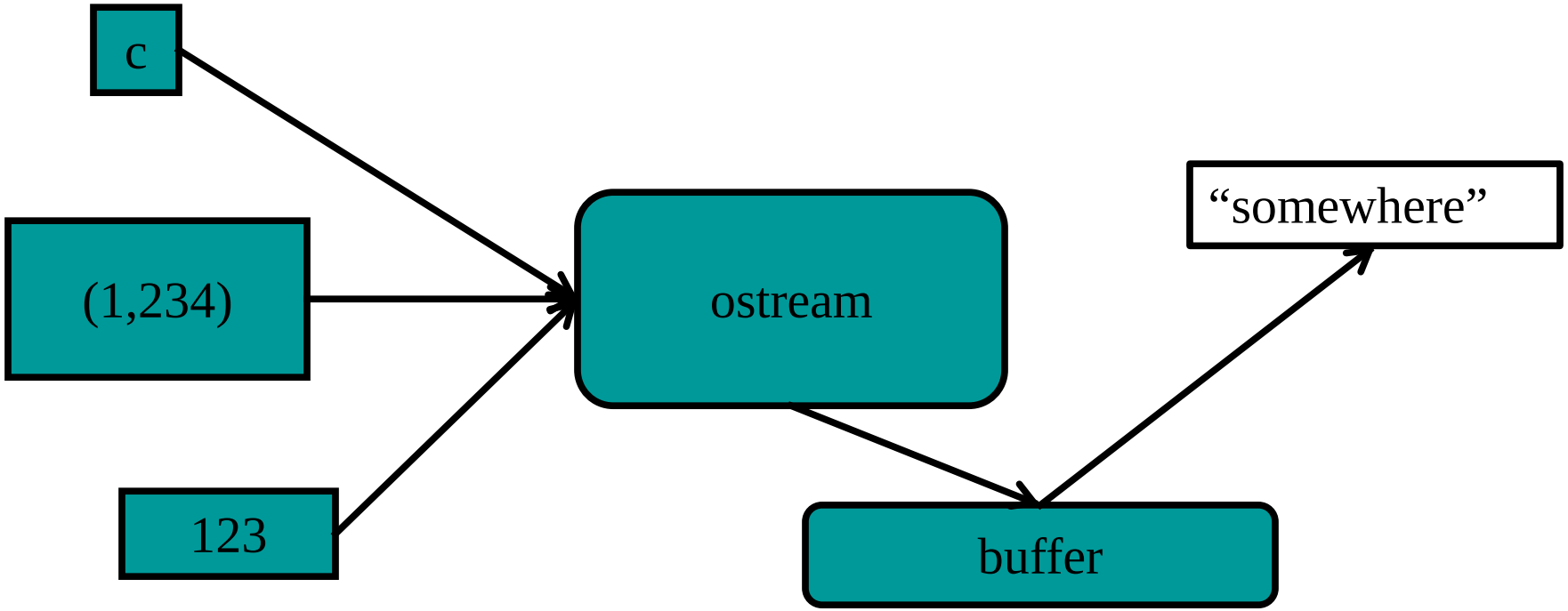
Input and Output

data source:



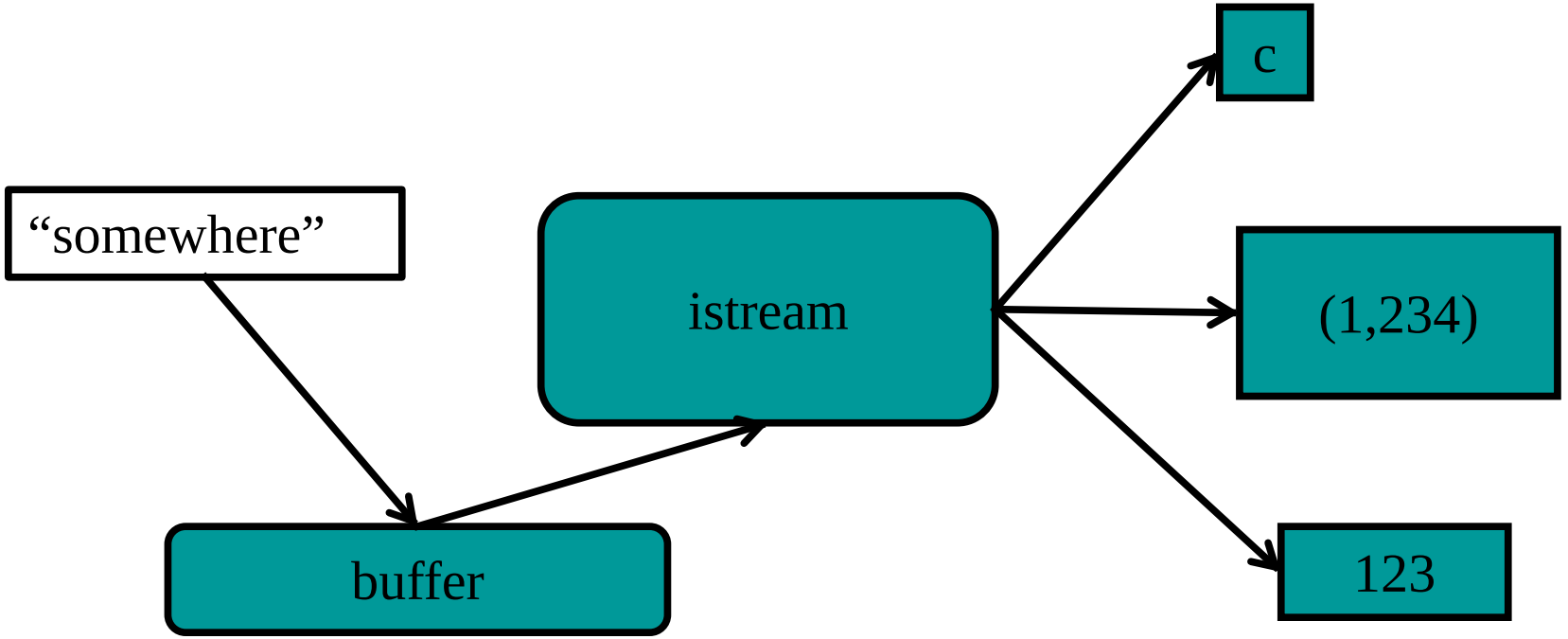
The stream model

- 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

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 (including format error)*
  - **bad()** *// something unexpected and serious happened*

- Note that good() and bad() are **not** opposites, despite the name
  - Name selection for these methods was especially tortured...

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 (Mac* / Linux)
  - State is **eof()**
- Something really bad
  - Disk format error
  - State is **bad()**

* A second Control-D at the start of a line may also be required

Reading a single value

- (At least) 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)

```
// 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
```

Characters

- You can also read individual characters

```
for (char ch; cin>>ch; ) {   // read into ch, skip whitespace characters
          if (isalpha(ch)) {
                              // do something
          }
}

for (char ch; cin.get(ch); ) {        // read into ch, handle whitespace separately
          if (isspace(ch)) {
                              // do something
          }
          else if (isalpha(ch)) {
                              // do something else
          }
}
```

Character classification functions

- If you use character input, you often need one or more of these (from header **<cctype>**):

–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')*
–isupper(c) *// is c an upper case letter?*
–islower(c) *// is c a lower case letter?*
–isalnum(c) *// is c a letter or a decimal digit?*

etc.

http://cplusplus.com/reference/cctype/

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
                throw runtime_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?

- 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";
        }
}
```

Output formats

- Integer values
  - **1234** (decimal)
  - **2322** (octal)
  - **4d2** (hexadecimal)
- Floating point values
  - **1234.57** (general)
  - **1.2345678e+03** (scientific)
  - **1234.567890** (fixed)
- Precision (for floating-point values)
  - **1234.57** (precision 6)
  - **1234.6** (precision 5)
- Fields
  - **\|12\|** (default for | followed by **12** followed by |)
  - **\| 12\|** (**12** in a field of 4 characters)

Numerical Base Output

dec hex oct

- You can change "base"
  - Base 10 == decimal; digits: 0 1 2 3 4 5 6 7 8 9
  - Base 8 == octal; digits: 0 1 2 3 4 5 6 7
  - Base 16 == hexadecimal; digits: 0 1 2 3 4 5 6 7 8 9 a b c d e f

```
// simple test:
    cout << dec << 1234 << "\t(decimal)\n"
         << hex << 1234 << "\t(hexadecimal)\n"
         << oct << 1234 << "\t(octal)\n";
// The '\t' character is a "tab"
```

- Results

1234 (decimal)
4d2 (hexadecimal)
2322 (octal)

“Sticky” Manipulators

- You can change “base”
  - Base 10 == decimal; digits: 0 1 2 3 4 5 6 7 8 9
  - Base 8 == octal; digits: 0 1 2 3 4 5 6 7
  - Base 16 == hexadecimal; digits: 0 1 2 3 4 5 6 7 8 9 a b c d e f

```
// simple test:
    cout << 1234 << '\t'
         << hex << 1234 << '\t'
         << oct << 1234 << '\n';
    cout << 1234 << '\n'; // the octal base is still in effect
```

- Results

1234 4d2 2322
2322

Most manipulators are “sticky”, and remain in effect until changes. A few are transient, and only affect the next output.

Other Manipulators

showbase noshowbase

- You can change "base"
  - Base 10 == decimal; digits: 0 1 2 3 4 5 6 7 8 9
  - Base 8 == octal; digits: 0 1 2 3 4 5 6 7
  - Base 16 == hexadecimal; digits: 0 1 2 3 4 5 6 7 8 9 a b c d e f

```
// simple test:
    cout << 1234 << '\t'
         << hex << 1234 << '\t'
         << oct << 1234 << endl;
    cout << showbase << dec;     // show bases via prefix
    cout << 1234 << '\t'
         << hex << 1234 << '\t'
         << oct << 1234 << '\n';
```

- The opposite of showbase is noshowbase
- Results

```
1234    4d2     2322
1234    0x4d2   02322
```

hex octal

Floating-point Manipulators

defaultfloat scientific fixed

- You can change floating-point output format
  - **defaultfloat** – **iostream** chooses best format using **n** digits (default)
  - **scientific** – one digit before the decimal point plus exponent; **n** digits after **.**
  - **fixed** – no exponent; **n** digits after the decimal point

```
// simple test:
cout << 1234.56789 << "\t(defaultfloat)\n"
     << fixed << 1234.56789 << "\t(fixed)\n"
     << scientific << 1234.56789 << "\t(scientific)\n";
```

- Results

1234.57 (defaultfloat)
1234.567890 (fixed)
1.234568e+03 (scientific)

Precision Manipulator

setprecision(digits)

- Precision (the default is 6) from <iomanip>
  - **defaultfloat** – precision is the number of digits
  - **scientific** – precision is the number of digits after the . (dot)
  - **fixed** – precision is the number of digits after the . (dot)

```
// example:
    cout << 1234.56789 << '\t' << fixed << 1234.56789 << '\t'
         << scientific << 1234.56789 << '\n';
    cout << general << setprecision(5)
         << 1234.56789 << '\t' << fixed << 1234.56789 << '\t'
         << scientific << 1234.56789 << '\n';
    cout << general << setprecision(8)
         << 1234.56789 << '\t' << fixed << 1234.56789 << '\t'
         << scientific << 1234.56789 << '\n';
```

- Results (note the rounding):

1234.57	**1234.567890**	**1.234568e+03**
1234.6	**1234.56789**	**1.23457e+03**
1234.5679	**1234.56789000**	**1.23456789e+03**

Output field width

setw(min_width)

- Width is the number of characters to be used for the next output operation
  - **Beware:** width is transient and applies to next output only (it doesn't "stick" like precision, base, and floating-point format)
  - **Beware:** output is never truncated to fit into field
    - (better a bad format than a bad value)

```
#include <iomanip>
    cout << 123456 <<'|'<< setw(4) << 123456 << '|'
         << setw(8) << 123456 << '|' << 123456 << "|\n";
    cout << 1234.56 <<'|'<< setw(4) << 1234.56 << '|'
         << setw(8) << 1234.56 << '|' << 1234.56 << "|\n";
    cout << "asdfgh" <<'|'<< setw(4) << "asdfgh" << '|'
         << setw(8) << "asdfgh" << '|' << "asdfgh" << "|\n";
```

- Results

```
123456|123456|  123456|123456|
1234.56|1234.56| 1234.56|1234.56|
asdfgh|asdfgh|  asdfgh|asdfgh|
```

Observation

fx Format flag manipulators (functions)

Independent flags (switch on):	
boolalpha	Alphanumerical bool values (function)
showbase	Show numerical base prefixes (function)
showpoint	Show decimal point (function)
showpos	Show positive signs (function)
skipws	Skip whitespaces (function)
unitbuf	Flush buffer after insertions (function)
uppercase	Generate upper-case letters (function)
Independent flags (switch off):	
noboolalpha	No alphanumerical bool values (function)
noshowbase	Do not show numerical base prefixes (function)
noshowpoint	Do not show decimal point (function)
noshowpos	Do not show positive signs (function)
noskipws	Do not skip whitespaces (function)
nounitbuf	Do not force flushes after insertions (function)
nouppercase	Do not generate upper case letters (function)
Numerical base format flags ("basefield" flags):	
dec	Use decimal base (function)
hex	Use hexadecimal base (function)
oct	Use octal base (function)
Floating-point format flags ("floatfield" flags):	
fixed	Use fixed floating-point notation (function)
scientific	Use scientific floating-point notation (function)
Adustment format flags ("adjustfield" flags):	
internal	Adjust field by inserting characters at an internal position (function)
left	Adjust output to the left (function)
right	Adjust output to the right (function)

This kind of detail is why you need (online) manuals – try this one:
http://www.cplusplus.com/reference/ios/

String streams

A **stringstream** (from <sstream>) reads/writes from/to a **string** rather than a file or a keyboard/screen.

This adds all stream capabilities to your string editing arsenal

```
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");         // will call error()
```

http://www.cplusplus.com/reference/sstream/stringstream/

String streams

- See textbook, cplusplus.com, or Stack Overflow 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 whitespace-terminated string

```
string name;
cin >> name;                // input: Dennis Ritchie
cout << name << '\n';       // output: Dennis
```

- Read a line

```
string name;
getline(cin, name);         // input: Dennis Ritchie
cout << name << '\n';       // output: Dennis Ritchie

// now what? Maybe:

istringstream ss(name);
ss >> first_name;
ss >> second_name;
```

Line-oriented input

- Prefer >> to **getline()**
  - i.e. avoid line-oriented input when you can

- People often use **getline()** because they see no alternative
  - But it easily gets messy
  - When trying to use **getline()**, you often end up
    - using >> to parse the line from a **stringstream**
    - using **get()** to read individual characters

```
int a, b;
while (infile >> a >> b)
{
    // process pair (a,b)
}
```

```
std::string line;
while (std::getline(infile, line))
{
    std::istringstream iss(line);
    int a, b;
    if (!(iss >> a >> b)) { break; } // error

    // process pair (a,b)
}
```

Code from http://stackoverflow.com/questions/7868936/read-file-line-by-line
Used under "non-commercial use" clause of the license agreement.