Input/Output Streams

Based on materials by Bjarne Stroustrup

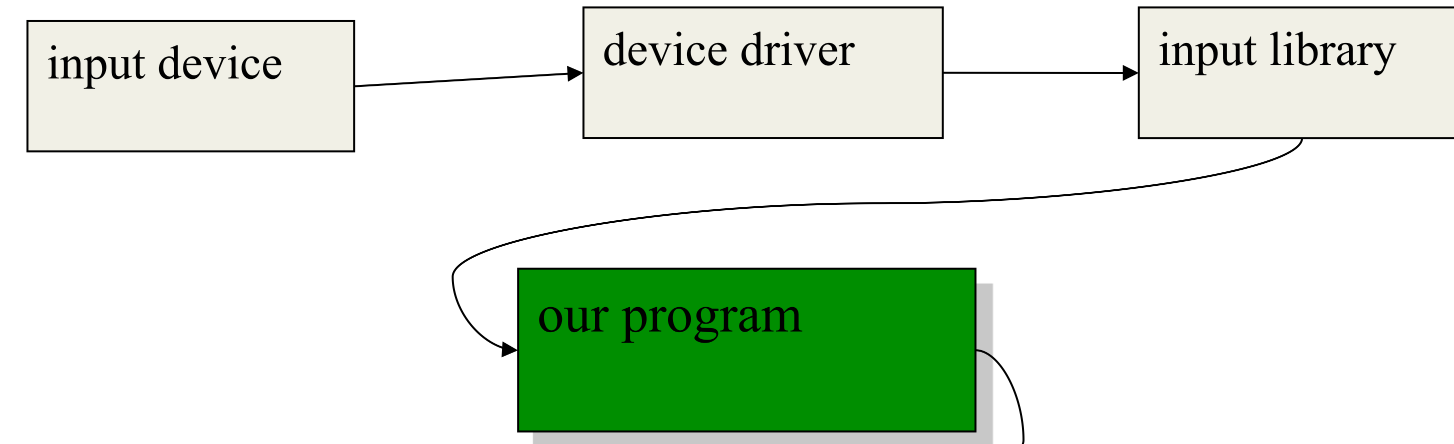
www.stroustrup.com/Programming

Overview

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

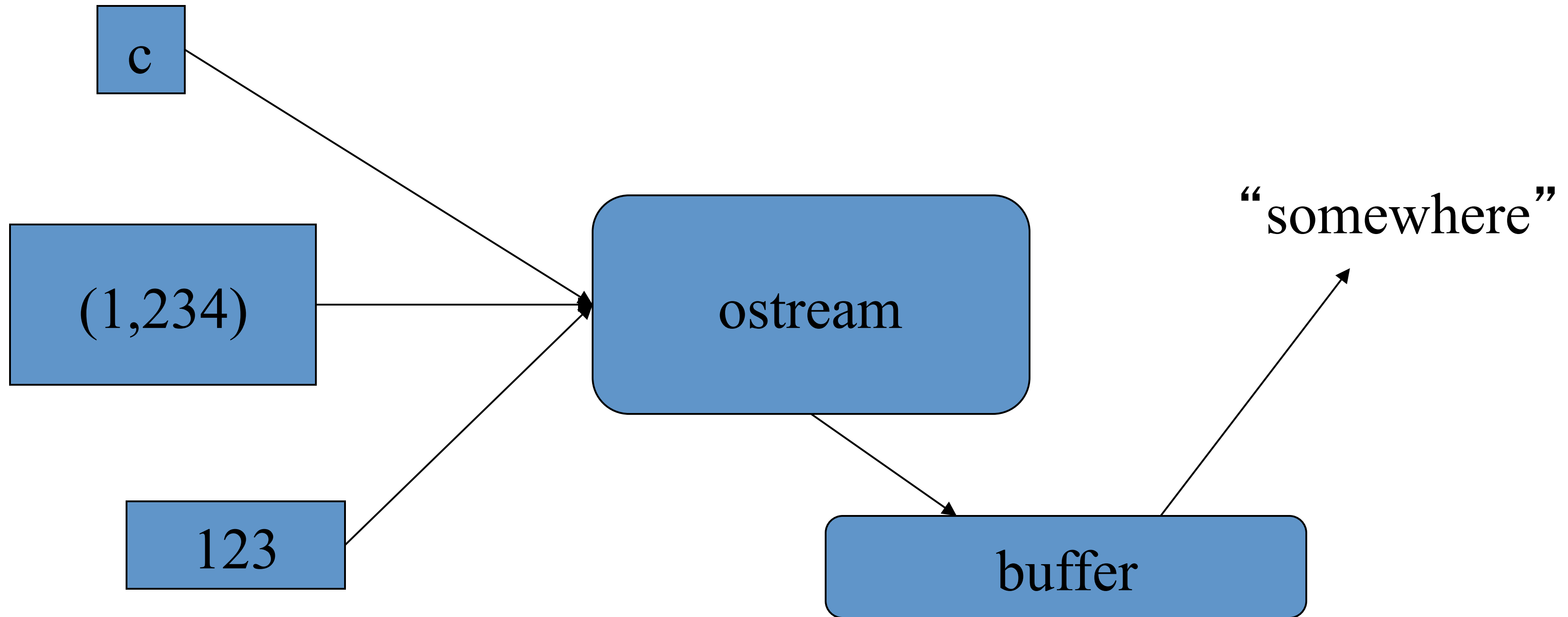
Input and Output

data source:



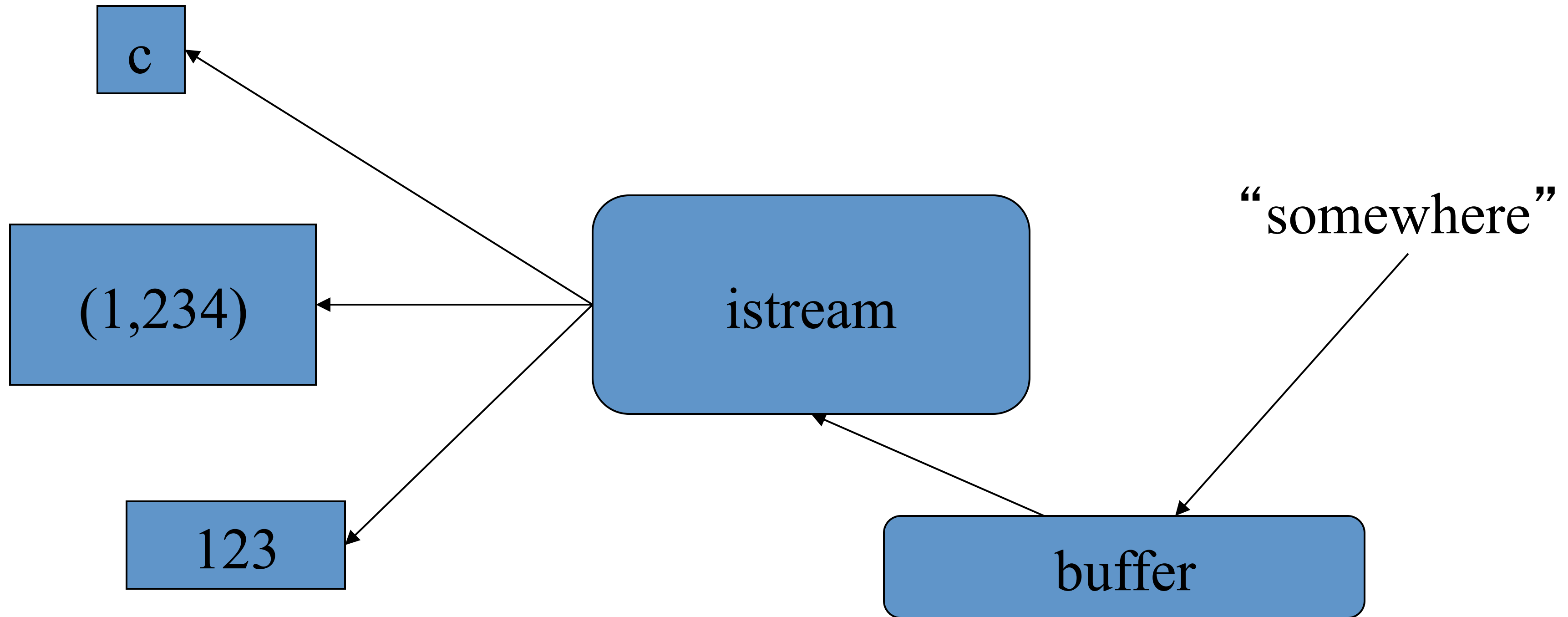
data destination:

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

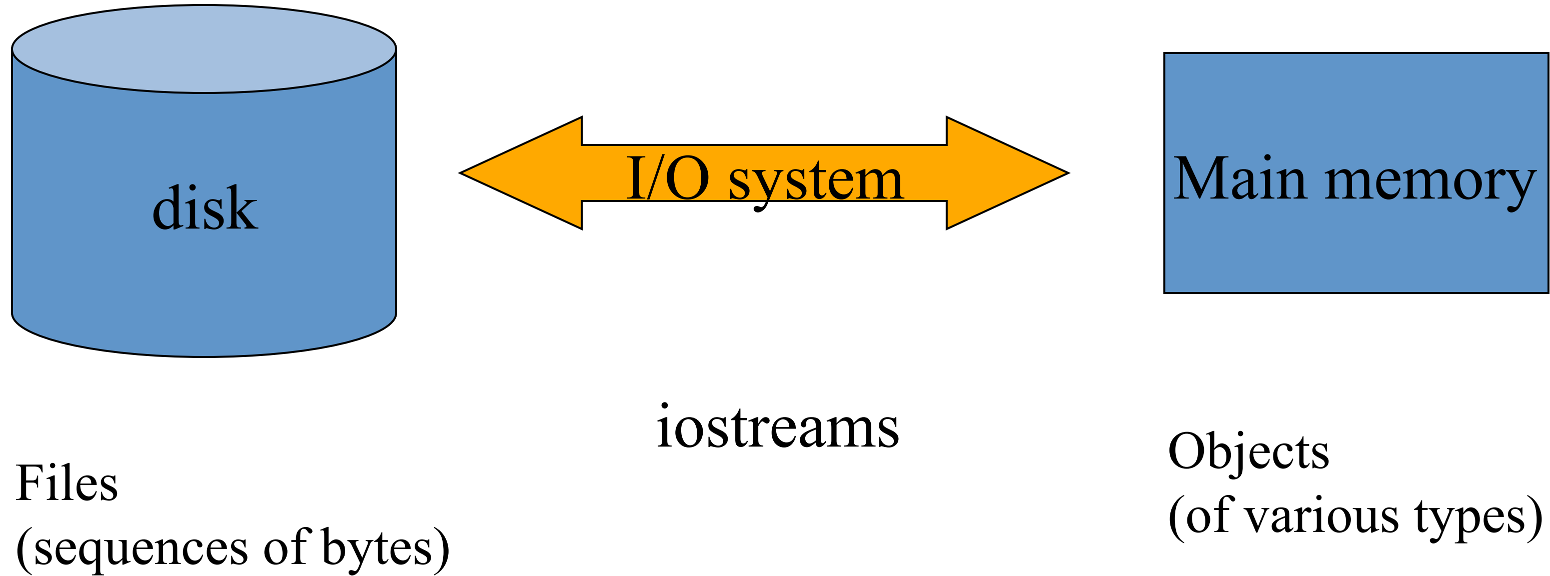
- We turn our computers on and off
  - The contents of our main memory is transient
- We like to keep our data
  - So we keep what we want to preserve on disks and similar permanent storage
- A file is a sequence of bytes stored in permanent storage
  - A file has a name
  - The data on a file has a 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

- 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

```
// …
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() gives a low-level ("system"
                        // or C-style) string from a C++ string

                        // defining an ifstream with a name string
                        // opens the file of that name for reading
    if (!ist) error("can't open input file ", name);
    // ...
```

Opening a file for writing

```
    // …
    cout << "Please enter name of output file: ";
    cin >> name;
    ofstream ofs(name.c_str()); // 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 ", name);
    // …
}
```

Reading from a File

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())  {
    while ( myfile.good() ) {
       getline (myfile,line);
       cout << line << endl;
    }
    myfile.close();
  }
  else
      cout << "Unable to open file";
  return 0;
}
```

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

```
struct Reading {   // a temperature reading
  int hour;        // hour after midnight [0:23]
  double temperature;
  Reading(int h, double t) :hour(h), temperature(t) { }
};

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
    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 (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()

```
// 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

```
void fill_vector(istream& ist, vector<int>& v, char terminator)
{     // read integers from ist into v until we reach eof() or terminator
    int i = 0;
    while (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

// 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";
}
```

- 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 (n==0) {   // Spot the bug!
      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";
        char ch;
        while (cin>>ch && !isdigit(ch)) ;      // throw away 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 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
      char ch;
      while (cin>>ch) {       // throw away non-digits
            if (isdigit(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?

```
// 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
}
```

Extra Material

Output Formatting

Observation

- As programmers we prefer regularity and simplicity
  - But, our job is to meet people's expectations
- People are very fussy/particular/picky about the way their output looks
  - They often have good reasons to be
  - Convention/tradition rules
    - What does 123,456 mean?
    - What does (123) mean?
  - The world (of output formats) is weirder than you could possibly imagine

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

- 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 "tab" (short for "tabulation character")

// 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
```

Other 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 << endl;        // '\n'
    cout << showbase << dec;  // show bases
    cout << 1234 << '\t'
            << hex << 1234 << '\t'
            << oct << 1234 << '\n';
// results:
    1234  4d2 2322
    1234  0x4d2    02322
```

Floating-point Manipulators

- You can change floating-point output format
  - general – **iostream** chooses best format using **n** digits (this is the 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\t(general)\n"    //  \t\t  to line up columns
     << fixed << 1234.56789 << "\t(fixed)\n"
     << scientific << 1234.56789 << "\t(scientific)\n";

// results:
1234.57         (general)
1234.567890     (fixed)
1.234568e+003   (scientific)
```

Precision Manipulator

- Precision (the default is 6)
  - general – precision is the number of digits
    - Note: the **general** manipulator is not standard, just in std_lib_facilities.h
  - **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+003
1234.6      1234.56789    1.23457e+003
1234.5679       1234.56789000     1.23456789e+003
```

Output field width

- A width is the number of characters to be used for the next output operation
  - Beware: width 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)

```
// example:
    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|
```

Extra Material

Files Modes

A file

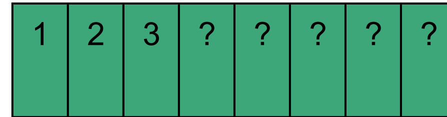
- At the fundamental level, a file is a sequence of bytes numbered from 0 upwards
- Other notions can be supplied by programs that interpret a “file format”
  - For example, the 6 bytes "123.45" might be interpreted as the floating-point number 123.45

File open modes

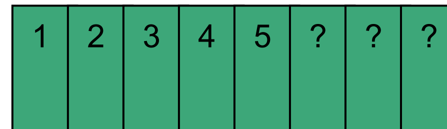
- By default, an **ifstream** opens its file for reading
- By default, an **ofstream** opens its file for writing.
- Alternatives:
  - **ios_base::app** *// append (i.e., add to the end of the file)*
  - **ios_base::ate** *// "at end" (open and seek to end)*
  - **ios_base::binary** *// binary mode – 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:

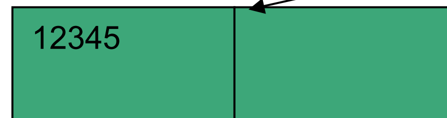


12345 as characters:



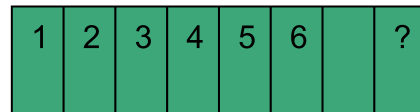
123 as binary:

12345 as binary:



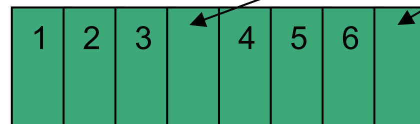
In binary files, we use sizes to delimit values

123456 as characters:



In text files, we use separation/termination characters

123 456 as 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

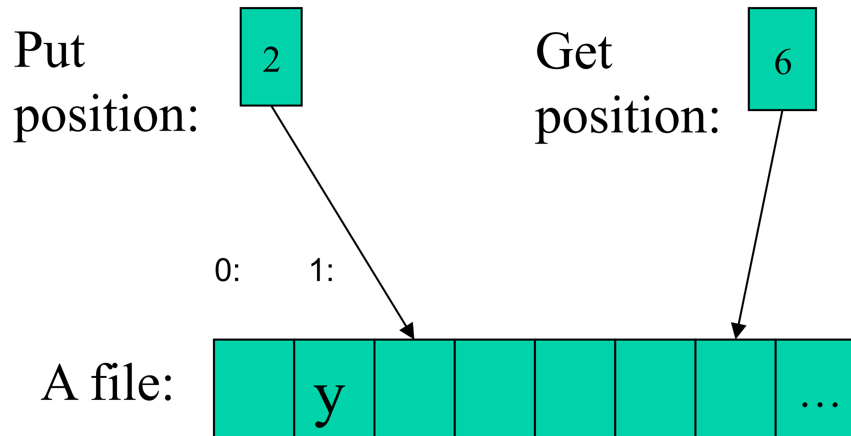
Binary files

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

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

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

Binary files

```
    vector<int> v;

    // read from binary file:
    int i;
    while (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
```

Positioning in a filestream




```
fstream fs(name.c_str());        // open for input and output

// …

fs.seekg(5);  // move reading position ('g' for 'get') to 5 (the 6th character)
char ch;
fs>>ch;       // read and increment reading position
cout << "character[6] 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
```

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

Extra Material

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");
    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()
```

String streams

- 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;            // 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;
```

Characters

- You can also read individual characters

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

while (cin.get(ch)) {   // read into ch, don't skip whitespace characters
    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?*

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 often 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