Customizing I/O

Overview

- Input and output
- Numeric output
  - Integer and Floating point
  - Numerical bases
  - Manipulators
- File modes
  - Binary I/O
  - Positioning
- String streams
- Line-oriented input
  - Character input
  - Character classification

Kinds of I/O

- Individual values
- Streams
- Textual
  - Type driven, formatted
  - Line oriented
  - Individual characters
- Numeric
  - Integer
  - Floating point
  - User-defined types

Observation

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

Output formats

- Integer values
  - **1234** (decimal)
  - **2322** (octal)
  - **4d2** (hexadecimal)
- Floating point values
  - **1234.56** (general)
  - **1.2345678e+03** (scientific)
  - **1234.567890** (fixed)
- Precision (for floating-point values)
  - **1234.56** (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
#include <iomanip>
    cout << dec << 1234 << "\t (decimal)"<<endl
         << hex << 1234 << "\t (hexadecimal)"<<endl
         << oct  << 1234 << "\t (octal)”<<endl;
// The '\t' character is “tab” (short for “tabulation character”)

// results:
    1234    (decimal)
    4d2     (hexadecimal)
    2322    (octal)
```

“Sticky” Manipulators

```
// simple test
cout << 1234 << '\t'
     << hex << 1234 << '\t'
     << oct  << 1234 << endl;

cout << 1234 << endl;    // the oct base is still in effect

// results:
   1234      4d2      2322
   2322
```

Other Manipulators

```
// simple test:
cout << 1234 << '\t'
     << hex << 1234 << '\t'
     << oct  << 1234 << endl;

cout << showbase << dec;     // show bases

cout << 1234 << '\t'
     << hex << 1234 << '\t'
     << oct  << 1234 << endl;

// results:
   1234      4d2      2322
   1234   0x4d2    O2322
```

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 **“.”** (dot)
  - **fixed** – no exponent, **n** digits after the decimal point

```
// simple test
cout << 1234.56789 << "\t\t (general) \n"    //  \t\t  to aline 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)
  - **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 << setprecision(5)
     << 1234.56789 << '\t' << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
cout << 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 wrong 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|
```

Observation

- This kind of details is what for you need textbooks, manuals, references, web search etc.
  - You **always** forget some of details when you need them :-)

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, 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.
- Modes (alternatives) (ios_base class enumerators):
  - **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. OS specific)*
  - **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(name1, ios_base::app);** *// append*
    // rather then overwrite
  - **fstream fs(name, ios_base::in | ios_base::out);** *// both*
    // in and out

Text vs binary files

123 as characters:

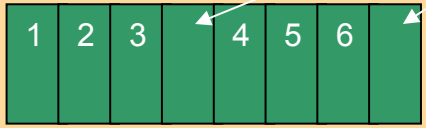
12345 as characters:

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 files

- 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 only when you must
  - E.g. image files, sound files, big data sets

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) cout<<"can't open input file "<< name<<endl;

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

    // "binary" tells the stream do not to try any manipulations with bytes
```

Binary files (cont'd)

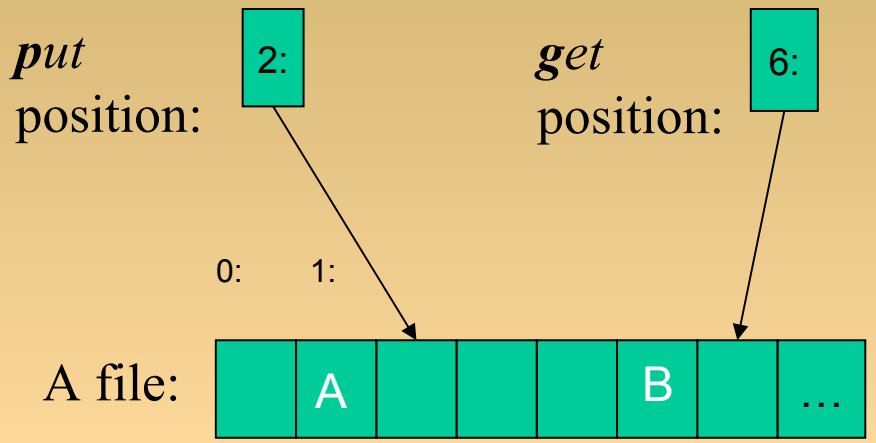
```
vector<int> v;
int i;
// read from binary file using bool ifstream::read(char*, int)
while (ifs.read((char*) &i, sizeof(int)))    // read 4 bytes into i
        v.push_back(i);

// … do something with v …

// write v to binary file using bool ofstream::write(char*,int)
for(int i=0; i < v.size(); ++i)
      ofs.write((char*) &v[i], sizeof(int));   // write 4 bytes from v[i]
```

```
// Generally, read / write has 2 arguments: address in memory (as
// char*) and number of bytes to read / write
```

Positioning in a file stream




```
fstream fs(name.c_str());     // open for input and output
// …
fs.seekg(5);    // move reading (get) position to 5 (the 6th character)
char ch;
fs >> ch;          // read character (1 byte) and increment reading position
cout << "character 6 is " << ch << '(' << int(ch) << ")"<<endl; //  B (66)
fs.seekp(1);      // move writing (put) position to 1 (the 2nd character)
fs << 'A';          // 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. Use it only if you really need (e.g. work with random access files on disk / in memory)
  - Handling of the end-of-file position is system dependent and basically unchecked

String streams

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

```
// function to convert characters in string s to floating-point value
double str_to_double(string s)
{
    istringstream is(s);  // make a stream so that we can read from s
    double d = 0.;
    is >> d;              // read from string into double
    if (!is) cout<<"double format error"<<endl;
    return d;
}

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

String streams

- Very useful for
  - formatting into a fixed-sized space (e.g. for GUI)
  - for extracting typed objects out of a string

Read string vs Read line

- Read a string

```
string name;
cin >> name;              // keyboard input: Dennis Ritchie
cout << name << '\n';     // output: Dennis ('till first white-space)
```

- Read a line

```
string line; string first_name; string second_name;
getline(cin, line);       // keyboard input: Dennis Ritchie
cout << line << endl;     // output: Dennis Ritchie
// parse this line
istringstream ss(line);
ss >> first_name;
ss >> second_name;
```

  - Better solution:

```
cin >> first_name >> second_name;  // do the same
```

Characters

- You can also read individual characters

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

while (cin.get(ch)) {  // read into ch, don't skip whitespace characters
    if (isspace(ch)) {    // is it space?
        // do something
    } else if (isalpha(ch)) {  // character?
        // do something else
    }
}
```

Character classification functions

- If you use character input, you often need one or more of these functions
- (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 with
  - usage of **>>** to parse the line from a **stringstream**
  - or usage of **get()** to read individual characters

Next talk

- Streams
- I/O errors
- User defined “<<” and “>>” operators

Practical part for today

- Write one base class and few derived classes.
  - Put declaration of classes in different header files.
- Create objects of derived classes via “new”.
- In the main() demonstrate RTTI: i.e. having pointer to base class, recognize on which derived class this pointer points to.

Practical part for today

- Write a function which converts “double” variable into a string, with characters representing this number.
  i.e. double v = 1234.56 should be converted to “1234.56”