

DM560
Introduction to Programming in C++

Input/Output Streams: Customizing

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[Based on slides by Bjarne Stroustrup]

Outline

1. Formatting
2. Files
3. Positioning
4. String Streams

Overview

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

Kinds of I/O

- Individual values (Chapters 4, 10)
- Streams (Chapters 10-11)
- Graphics and GUI (Chapters 12-16)
- Text
 - 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 about the way their output looks
They often have good reasons to be:
 - Convention/tradition rules
 - What does 110 mean?
 - What does 123,456 mean?
 - What does (123) mean?
 - The world (of output formats) is weirder than you could possibly imagine

Outline

1. Formatting

2. Files

3. Positioning

4. String Streams

Output Formats

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

Numerical Base Output

You can change [base](#):

| Base | Name | Digits: |
|------|-------------|---------------------------------|
| 10 | decimal | 0 1 2 3 4 5 6 7 8 9 |
| 8 | octal | 0 1 2 3 4 5 6 7 |
| 16 | hexadecimal | 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)
```


Base Manipulators

“Sticky” manipulators

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

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

Other manipulators

```
// 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

| | |
|---------------------|---|
| defaultfloat | <code>iostream</code> chooses best format using <code>n</code> digits (this is the default) |
| fixed | no exponent; <code>n</code> digits after the decimal point |
| scientific | one digit before the decimal point plus exponent; <code>n</code> digits after |

```
// simple test:
cout << 1234.56789 << "\t\t(defaultfloat)\n"      // \t\t to line up columns
    << 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

Precision (the default is 6)

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

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|
```

Obervation

~> This kind of detail is what you need textbooks, manuals, references, online support, etc. for.
You *always* forget some of the details when you need them

Outline

1. Formatting
2. Files
3. Positioning
4. String Streams

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., 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

- If **123** is stored as an integer (ie, a binary number) it occupies 4 bytes.
- If **‘123’** is stored as a string it occupies 3 characters (actually 4). Chars have variable length but it simplifies to think they occupy one byte.

123 as characters:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | ? | ? | ? | ? | ? |
|---|---|---|---|---|---|---|---|

12345 as characters:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | ? | ? | ? |
|---|---|---|---|---|---|---|---|

123 as binary:

| | |
|-----|--|
| 123 | |
|-----|--|

12345 as binary:

| | |
|-------|--|
| 12345 | |
|-------|--|

123456 as characters:

| | | | | | | | |
|---|---|---|---|---|---|--|---|
| 1 | 2 | 3 | 4 | 5 | 6 | | ? |
|---|---|---|---|---|---|--|---|

123 456 as characters:

| | | | | | | | |
|---|---|---|--|---|---|---|--|
| 1 | 2 | 3 | | 4 | 5 | 6 | |
|---|---|---|--|---|---|---|--|

- 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

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);

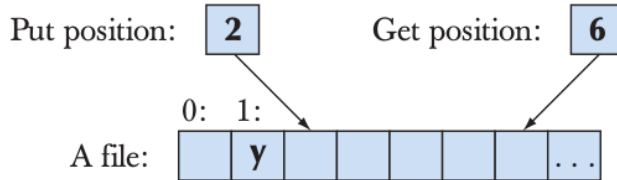
    cout << "Please enter output file name\n";
    string oname;
    cin >> oname;
    ofstream ofs {oname, ios_base::binary};          // note: binary
    if (!ofs) error("can't open output file ", oname);

    vector<int> v;    // read from binary bytes from file:
    for (int i; ifs.read(as_bytes(i), sizeof(int)); )    v.push_back(i);
    //...
    for(int i=0; i<v.size(); ++i)
        ofs.write(as_bytes(v[i]), sizeof(int));        // note: writing binary bytes
    return 0;
}
// For now, treat as_bytes() as a primitive
// Warning! Beware transferring between different systems
```

Outline

1. Formatting
2. Files
3. Positioning
4. String Streams

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

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

Outline

1. Formatting
2. Files
3. Positioning
4. 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"); // will call error()
```

- Document yourself on `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;           // 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

```
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, 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 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

C++14

- Binary literals

`0b1010100100000011`

- Digit separators

`0b1010'1001'0000'0011`

Can also be used for decimal, octal, and hexadecimal numbers

- User-Defined Literals (UDLs) in the standard library

- Time: `2h+10m+12s+123ms+3456ns`

- Complex: `2+4i`

Summary

1. Formatting
2. Files
3. Positioning
4. String Streams