

The IO-Library

Michael Hanke

School of Engineering Sciences

Program construction in C++ for Scientific Computing



- ① Introduction
- ② Basics About Streams
- ③ Formatted And Unformatted I/O
- ④ A Comprehensive Example
- ⑤ Summary

We have already used some I/O-functionality in C++:

- `cin`, `cout`, `cerr` objects which read and write the standard channels, respectively.
- `cerr` is typically used to write error or debug messages.
- The `<<` and `>>` operators which are used to read input or to write output.
- The header files `<iostream>` and `<iomanip>` containing the declarations of the classes and standard objects.
- Simple attributes like `endl` or formatting.

What More is Available?

Introduction

Basics About
StreamsFormatted
And
Unformatted
I/OA Comprehen-
sive
Example

Summary

- iostream** Classes for describing character streams. Contains, among others, the classes `istream` and `ostream`.
- fstream** Classes for reading and writing from/to a file. Contains, among others, the classes `ifstream` and `ofstream`.
- sstream** Classes for reading and writing from/to a string. Contains, among others, the classes `istringstream`, `ostringstream`, `stringstream`.
- Operators for formatted I/O: `<<`, `>>`
 - Member functions for unformatted I/O
 - Functions for querying and setting the state of a stream (eg, `eof()`)

Be Careful

- I/O objects cannot be copied.
- I/O objects cannot be assigned.

Condition States

Each stream owns a flag byte indicating the state(s) of that stream.

Flag	Meaning
failbit	The last I/O operation failed
eofbit	At an earlier reading attempt, the file was read behind its end
badbit	System level failure during I/O

Manipulating Flags

Function	Explanation
<code>void clear()</code>	Reset all flags
<code>bool good()</code>	Returns true if all flags are reset
<code>bool fail()</code>	Returns true if badbit or failbit is set
<code>bool eof()</code>	Returns true if eofbit is set
<code>bool bad()</code>	Returns true if badbit is set
<code>bool operator!()</code>	Returns fail()
<code>operator bool()</code>	Returns not fail()

Note: The programmer can also set flags by bit manipulations, eg:

```
cin.clear(cin.rdstate() & ~cin.failbit);
```

(resets failbit of cin)

Example: Wrong Usage of eof()

```
char c;  
while (!cin.eof()) {  
    cin.get(c);  
    // Do something  
}
```

Question: *What is wrong?*

Correct Usage of eof()

```
char c;  
while (true) {  
    cin.get(c);  
    if (cin.eof()) break;  
    // Do something  
}
```

Question: Why does the following version work?

```
char c;  
while (cin.get(c)) {  
    // Do something  
}
```

Flushing An Output Buffer

- For efficiency reasons, output streams use a buffer.
- This may become a problem if user input is required:

```
os << "Enter a value: ";  
is >> myvalue;
```

How can it be ensured that the prompt is written before the program waits for the input?

Conditions For Flushing the Buffer

- The program completes normally.
- If the buffer is completely filled, its contents will be flushed.
- The programmer can require flushing by using manipulators:

```
cout << "hi!" << endl;  
cout << "hi!" << flush;  
cout << "hi!" << ends; // for strings
```

- The `unitbuf` manipulator sets the stream to empty the buffer after each output operation (standard for `cerr!`).
- An output stream is tied to an input stream. Eg, `cout` will be flushed if `cin` will be read.

- Streams can be associated with a file. This accomplished by an `fstream` object.
- Basic operations:

Function	
<code><i>fstream</i> fs;</code>	Creates an unbound stream
<code><i>fstream</i> fs(s);</code>	Creates an <code>fstream</code> on bound file <code>s</code> (of <code>char*</code> or <code>string</code>)
<code><i>fstream</i> fs(s,mode);</code>	As before, but opening with a mode
<code>void fs.open(s)</code>	Binds the (text file) file <code>s</code> to <code>fs</code>
<code>void fs.open(s,mode)</code>	
<code>void fs.close()</code>	Closes <code>fs</code>
<code>bool fs.is_open()</code>	

Mode can be `in`, `out`, `app`, `trunc`, `ate`, `binary` (defined in class `ios`).

Example: A File Copy Program

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#define MAX_PATH_LEN 1024;
using namespace std;
```

Example (cont)

```
int main() {
    char c, fn[MAX_PATH_LEN];
    cout << "Source file name: ";
    cin >> setw(MAX_PATH_LEN) >> fn;
    ifstream f1(fn);
    if (!f1) {
        cerr << "Error while opening file!";
        exit(EXIT_FAILURE);
    }
    cout << "Destination file: ";
    cin >> setw(MAX_PATH_LEN) >> fn;
    ofstream f2(fn);
    if (!f2) {
        cerr << "Error creating file!";
        exit(EXIT_FAILURE);
    }
    while (f1.get(c)) f2.put(c);
}
```

Formatted I/O

- Unless the file mode binary is given, streams are considered to be text streams.
- The coding of the streams depends on the computer's localization (LATIN1, ISO8859-15, UTF-8 etc).
- Formatted I/O is easiest handled by using the operators << and >>.
- Manipulators can be used to change the format state.
- Format changing manipulators usually remain in effect for all subsequent I/O.
- When defining own classes, it is often a good idea to provide resonable replacements for << and >>.
- Since formatting is handled by manipulators, it is often a good idea not to explicitly use formatting in the own implementations.
- For details, consult the documentation.

Unformatted I/O

- Unformatted I/O considers the file as a sequence of bytes. So the corresponding built-in type is `char`.
- I/O is accomplished by member functions of the corresponding class.
- All these member functions return a reference to the stream object involved (with only a minor number of exceptions).
- Each stream has a marker indicating the actual position where the next byte is read from/put to.

When is unformatted I/O useful?

- Checkpointing in larger programs.
- Temporary files, eg, for out-of-memory algorithms.

Warning

Binary files are inherently system dependent. A common problem is to use files generated on big-endian machines on little-endian ones and vice versa!

Single-Byte Operations

Operation	Explanation
<code>is.get(c)</code>	Put the next byte from <code>is</code> in <code>char c</code>
<code>os.put(c)</code>	Put the <code>char c</code> onto <code>os</code>
<code>int is.get()</code>	Returns the next byte from <code>is</code>
<code>is.unget()</code>	Reduce the position marker by one
<code>int is.peek()</code>	Same as <code>is.get()</code> , but do not change the position pointer
<code>is.putback(c)</code>	Same as <code>is.unget()</code> but <code>c</code> must be the one last read

Note: `is.get()` and `is.peek()` can return an end-of-file marker:

```
while ((ch = cin.get()) != EOF) // Do something
```

Multi-Byte Operations

Operation	Explanation
<code>is.read(sink,size)</code>	Reads up to <code>size</code> bytes from <code>is</code> into <code>sink</code>
<code>streamsize is.gcount()</code>	Returns numbers of bytes read by the last unformatted
<code>os.write(source,size)</code>	Writes <code>size</code> bytes from <code>source</code> to <code>os</code>
<code>is.ignore(size=1,delim=EOF)</code>	Reads and ignores at most <code>size</code> bytes up to and including

Multi-Byte Operations (cont)

- `is.get(sink,size,delim)` and `is.getline(sink,size,delim)` read bytes until one of the following conditions is met:
 - `size-1` bytes are read;
 - End-of-file is encountered;
 - The `delim` character is encountered.
- `getline(sink,size,delim)` reads `delim` and discards it.
- `get(sink,size,delim)` lets `delim` on the input stream!

File Direct Access

- The position marker for indicating the next I/O position can be manipulated:

Operation	Explanation
<code>streamoff is.tellg()</code> <code>streamoff os.tellp()</code>	Returns the marker position
<code>is.seekg(pos)</code> <code>os.seekp(pos)</code>	Sets the marker position
<code>is.seekg(off,dir)</code> <code>os.seekp(off,dir)</code>	Move the marker off positions from the position defined by <code>dir</code> (beg, end, cur)

- The g-versions should be used for istreams, while the p-versions for o-streams.
- Obviously, using these functions is only meaningful for file or string I/O.

Remark

- For binary files (and direct access), it might be worth to consider the C counterparts.
- However, **do not mix C++ and C output to the same files!**

Given a class:

```
class Matrix {  
    int m_, n_;  
    double *x_;  
public:  
    Matrix() : m_(0), n_(0) {}  
    dump(char *file) const;  
};
```

Write a function to dump the matrix to a file!

```
#include <iostream>
Matrix::dump(char *file) const {
    ofstream f(file, ofstream::out | ofstream::binary);
    if (!f) {
        cerr << "Error creating file!";
        exit(EXIT_FAILURE);
    }
    f.write(&m_, sizeof(int));
    if (!f){
        cerr << "Error writing to file!";
        exit(EXIT_FAILURE);
    }
    f.write(&n_, sizeof(int));
    if (!f){
        cerr << "Error writing to file!";
        exit(EXIT_FAILURE);
    }
    f.write(x_, m_*n_*sizeof(double));
    if (!f){
        cerr << "Error writing to file!";
        exit(EXIT_FAILURE);
    }
    f.close();
}
```

Summary

- Some more details about C++ I/O
- What comes next:
 - Move, copy, domains