

GNIRTS ESAC REWOL

Bringing UNIX filters to iostream

Jan Christiaan (JC) van Winkel - <jc@atcomputing.nl>
John van Krieken - <john@atcomputing.nl>
AT Computing

1. Introduction

Iostream has proved to be a versatile and expandable system for I/O. It is very easy to add the ability to output and read object types. Where

```
Matrix mymat;  
// ....  
printf("%M", mymat);
```

would be very hard to accomplish in stdio, the equivalent:

```
Matrix mymat;  
// ....  
std::cout << mymat
```

just requires overloading the left shift operator for `std::ostream&` and `const Matrix&`. Also adding manipulators, even those that have to remember some state in `ios` is easy, through the `xalloc()` interface. However, adding new input and output streams to iostream remains some sort of black art. There are so many things you have to keep in mind, including the stream buffers.

Working in a UNIX environment, we are used to tinkering with the output of programs with filters to suit our needs. This experience prompted us to do a similar thing for iostream.

In this article, we present a framework through which adding a new `ostream` to iostream is as easy as writing one function object class. The scope of these new `ostreams` is limited - you can only alter the information going to the `ostream`. Nevertheless, we will give some useful applications.

2. UNIX filters

UNIX has a long standing tradition of openness and allowing users to manipulate the data that is read or written by programs. Data flows through the standard input and output channels. If you do not like the way a program presents its output, you can alter that output by having a program filter the data. The standard output channel of your program is connected to the standard input channel of the filter.

The notation for this connection is the vertical bar “|”, symbolizing the pipe through which the data flows. This way, data can be sorted, line numbers can be added or data can be filtered so only lines matching a specific regular expression will be output.

Some examples:

```
myprog | nl
```

to have line numbers added to every line of output of `myprog`

```
myprog | fgrep "abc"
```

to only let lines containing the text `abc` pass.

Of course, these can be combined:

```
myprog | fgrep "abc" | nl
```

will show the `abc` lines in a numbered fashion.

Note that these operations need not be commutative:

```
myprog | nl | fgrep "abc"
```

will show the `abc` lines with their line numbers as they were in the original output of `myprog`.

```
myprog | tee orig | nl | tee numbered | fgrep "abc"
```

will do the same, except that the intermediate results are copied by two T's in the pipe to the files named `orig` (which will hold the original output of `myprog`) and `numbered` (which will hold the line-numbered output of `myprog`).

3. Filtering streams

A few times when we required a new type of `ostream`, it was only to be able to slightly manipulate the output that was produced. Suppose you are using a library with functions that expect an `ostream` to write to. You want to alter the way the library outputs data, but you do not have the source code of the library. `ofiltstream` allows you to intercept and alter the data before it is sent out. Some examples for its use are:

- Being able to prepend every output line with a date and time stamp for logging purposes.
- Being able to also save all data output to cout to a file (via an ofstream).
- Being able to encrypt or compress output.

Here are a few examples to give you a taste of the possibilities:

```
logger logfilt; // a logging filter object
std::ofstream thefile("mydata.log");
ofiltstream logfile(logfilt, thefile); // a filtering ostream
logfile << "Hello World\n";
```

could cause the file mydata.log to contain something like:

```
Thu Jan 30 17:23:44 2003: Hello World
```

3.1 Implementation design

- For the sake of simplifying the discussion, the implementation shown here is not “char-traits-correct” and assumes narrow characters only. □

We will not discuss the setup of the iostream library. For understanding the inside view of our ofiltstream, we expect some understanding of the structure of iostream.

The implementation is done by creating a new class ofiltstream derived from std::ostream. This class uses a filtstreambuf class for the actual output of characters.

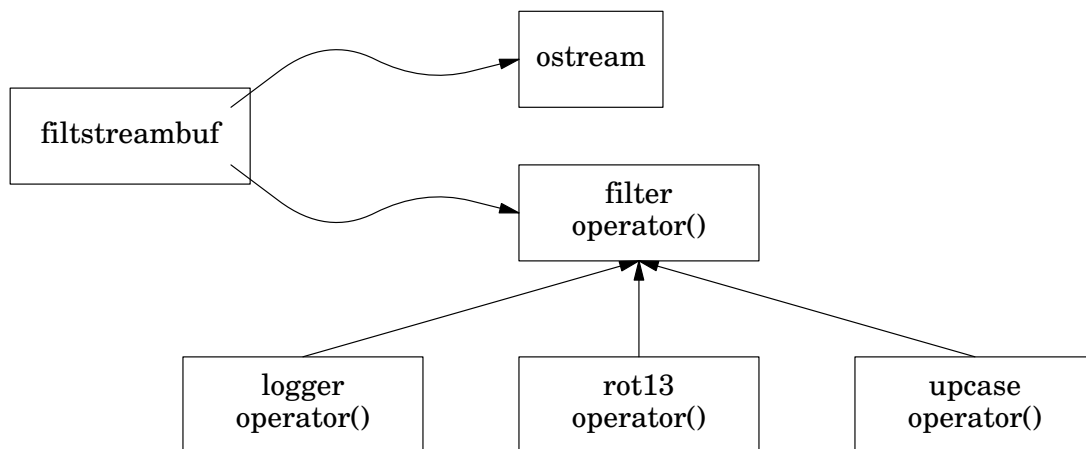


Figure 1. The filtstream structure

`filtstreambuf` is fairly simple (see figure 1). It disables all buffering. Hence, every call to `sputc` in `std::streambuf` will result in an `filtstreambuf::overflow()` call. This is where our `filtstreambuf` can do its

work: it calls the function object to output the character to the associated ostream.

The filter need not output one character per character input. It can suppress certain characters or output more than one character per character received. It can also buffer characters should this be needed. The return value reflects the state of the filter itself. In the case of simple character manipulating filters, this should always be `std::ios::goodbit`. The state of the ostream it outputs to may have gone bad, but that will be taken care of by the `filtstreambuf`: it combines the state of the embedded ostream with the state of the filtering object, and sets the state of the encapsulating `ofiltstream` with that.

```
// overflow will call associated function
// object and use its return value to set the
// state of the ostream the filtstreambuf belongs to
filtstreambuf::traits_type::int_type
filtstreambuf::overflow(traits_type::int_type c) {
    if( !traits_type::eq_int_type(c, traits_type::eof()) ){
        std::ios::iostate funcstate=(*thefunc)(thestream, c);
        boss.setstate(thestream.rdstate()|funcstate);
        return c;
    } else {
        return traits_type::not_eof(c);
    }
}
```

The core of the `ofiltstream` system is the stream buffer `filtstreambuf`. It has three data members: a reference to the original ostream, a reference to the ostream it is a part of, and a pointer to the filter object. Whenever the `overflow()` function is called, it will call the function object to manipulate the overflowing character, and output the data to the stream. The return value of the filter is bitwise or-ed with the ostream's state so is can set the state of the ostream it itself is part of. Hence, the `ofiltstream`'s state not only reflects the state of the underlying ostream, but also of the work the filter has done.

3.2 Different designs

In the very first setup we experimented with, the filtering objects accepted a character and returned a character. This has the advantage that the `filtstreambuf` and `ofiltstream` can be templated classes, so we use compile time (template based) polymorphism. No virtual functions are needed. The character-in character-out setup may be fine if all you want is to make a toupper filter or a rot13¹ filter, but if you need to change the number of characters that are output, this setup will not do. Therefore we quickly abandoned this setup in favor of

-
1. rot13 is an "encryption" technique that operates by replacing a lowercase or uppercase character by the character that is 13 characters higher or lower in the alphabet. So "a" becomes "n" and "n" becomes "a". This technique was (is?) used on USEnet to hide the punchline of jokes. Most news reader software has an option to rot13 text, though some people seem to have acquired the skill to read rot13-ed text directly.

three other setups:

- The filtering function objects get two parameters: the overflowing character, and the stream to output to. It returns an `ios::iostate` to indicate if it was able to handle the character. *Templates* are used for compile-time polymorphism.
- The filtering function objects get two parameters: the overflowing character, and the stream to output to. It returns an `ios::iostate` to indicate if it was able to handle the character. *Virtual functions* are used for run-time polymorphism.
- The filtering object communicates with the `streambuf` through `std::string` and lets the `filtstreambuf` do the output operation on the altered string. An exception of type `ios::iostate` is thrown by the filter if a problem occurs. `filtstreambuf` catches the exception and adjusts the stream's state according to the value of the exception.

We have tested and timed the template based design, but from a user's point of view, the performance loss due to virtual functions was minimal compared to the notational inconveniences. Also, implementing some filters proved difficult in the templated version. Therefore, we will not discuss this setup in much detail.

From an encapsulation point of view, the version using `string` is the cleanest design since it does not let the filtering object know about the stream. We have implemented both setups and came to the conclusion that though the `string` version is the cleanest, performance considerations make us prefer the `char/ostream&` version above the `char/string` version. We have some performance measurements in section 9.

4. Writing filtering classes

A filtering object is reasonably simple to make. All you have to make is a function object that takes a character and then outputs it to the stream. It should return an `ios::iostate` flag that tells whether or not the function was able to do its work. Note that `ofiltstream::overflow()` that called the filter will set its state as a combination of the state the output stream `os` below and the return value of the filter. Therefore, the return value of the filter does not need to represent the state of the stream it outputs to. As an example:

```
class upcase : public filter {
public:
    std::ios::iostate operator()(std::ostream& os, char c) {
        os << char(std::toupper(c));
        return std::ios::goodbit;
    }
};
```

The filtering objects may also buffer data if they so desire as is seen in this line-reversing filter:

```
class reverser : public filter {
public:
    std::ios::iostate operator()(std::ostream& os, char c) {
        if (c!='\n') {
            s.push_back(c);    // could throw
        } else {
            std::copy(s.rbegin(), s.rend(),
                    std::ostream_iterator<char>(os));

            os << c;
            s.clear();
        }
        return std::ios::goodbit;
    }
private:
    std::string s;
};
```

This filter reverses lines. It buffers characters in a string up to a newline, and then copies the string to the stream using reverse iterators.

A complication

A definition like

```
ofiltstream out(reverser(), std::cout);
```

will create a temporary filter object whose lifetime does not exceed the expression. Therefore, we must make sure that a *copy* is made. This suggests passing the filters to the `ofiltstream` constructor by value. However, this will hamper the polymorphic behavior of the filters. Since the `operator()` in the filters is a virtual functions, we cannot let the constructor take the parameters by value. This has been solved as follows:

All filters have to implement a `clone()` function. The implementation is simple (for `reverser`):

```
reverser::reverser* clone() const { return new reverser(*this); }
```

Since `clone()` uses the copy constructor, more complicated classes may have to implement their copy constructor as well.

As `clone()` is defined as a pure virtual function in the base class `filter`, the `ofiltstream` constructor can now make clones. Of course the newly allocated filter has to be deallocated as well. Fortunately the `filtstreambuf`'s destructor can take care of that.

5. Some simple filters

Several simple filters have been written:

`null`

A black hole. This filter will cause all output to be discarded.

`felix`

A simple `cat`. This filter just lets the data flow right through it without altering anything in the data flow. Useful for performance measurements.

`lowercase`

Transforms the data stream by converting all upper case characters to lower case.

`uppercase`

Transforms the data stream by converting all lower case characters to upper case.

`fgrep`

Filters on a line by line basis. Only lets those lines through that contain the substring² as passed to `fgrep`'s constructor.

`rot13`

Encrypts all upper and lower case characters using the good old shift-by-13-characters. When applied twice, the data passes through unaltered. Good for riddles and offensive jokes.

`logger`

Filters on a line by line basis. Every line will be printed with a date and time stamp in front of it.

`reverser`

Works on a line by line basis. Reverses the characters in the line.

`tee`

Works like `felix`, except that the constructor accepts a number of `ostream&`s. `tee` will copy the data to all these streams.

2. using the boost regular expression classes, this is easy to extend to a full-blown `grep` implementation that filters not on fixed strings, but on regular expressions.

6. Using the filters

The filters are used by creating a filter object, and then an `ofiltstream` that is attached to this filtering object and a pre-existing `ostream`:

```
reverser rev;
ofiltstream out(rev, std::cout);
// or
// ofiltstream out(reverser(), std::cout);

out << "Hello world!\n";
```

will output

```
!dlrow olleH
```

Since `out` above is also an `ostream`, it can itself be used with yet another filter object:

```
upcase upper;

ofiltstream myout(upper, out);

myout << "lower case string";
```

outputs GNIRTS ESAC REWOL.

7. Notational conveniences

In the previous section, we used two `ofiltstream` objects to define an “uppercasing” “reversing” filter. This is not very notationally convenient. If it would be possible to write:

```
reverser rev;
upcase upper;
ofiltstream myout(rev | upper, std::cout);
```

that would be a lot more like filtering à la UNIX.

Implementing this is obviously done by creating the function `filter operator|(filter, filter)`.

Here again we have the temporary problem: an expression like `rev|upper` will create a temporary variable. Fortunately, the `filter::clone()` function we needed for the constructor anyway helps us here too.

The `operator|(const filter&, const filter&)` returns a `combifilter` object³. This filter object stores pointers to the cloned two filters. Its

`operator>()()` first applies the first filter and then the second. The `combifilter` destructor takes care of deleting the cloned filters.

Summarizing, a filter has to implement:

- `operator()`
- `clone()`
- as usual, a copy constructor if the default copy constructor is not good enough.

Having written all this, we can make the following useful `ofiltstream`:

```
logger logfilt;
std::ostream dbg("out.debug");
ofiltstream out(tee(std::cout) | logfilt, dbg);
```

which will send output to `cout` and a time-stamped copy of that to the file `out.debug`.

8. Bringing Unix filters to iostream

UNIX lovers may be happy with something like

```
ofiltstream myout(rev | upper, std::cout);
```

but can we push the envelope even further? Something like:

```
unixfilter doit("awk '{ print $3 }'");
ofiltstream out(doit | log, cout);
// or:
// ofiltstream out(unixfilter("awk '{ print $3 }'") | log, cout);
```

to get date/time stamped lines where only the third word in these lines is printed.⁴

This can be done. But it is clear that the `unixfilter` class will be more complex than a simple `rot13` filter. Somehow, it will have to create two pipes (one to send the data to the UNIX command to, and one to get the UNIX command's output from). It also needs to create a child process using the UNIX `fork()` system call. This child first manipulates standard input, standard output and the pipes, so that the pipes become standard input and standard output of the shell command. When all that has been done, it can replace itself by a UNIX shell that will interpret the command. This is done using the `exec()` system call.

3. In the templated design, `operator|()` returns a `combifilter<T1, T2>`.

4. `awk` (named after the makers Aho, Weinberger and Kernighan) is a tool that allows easy manipulation of lines and columns of files (including calculating subtotals and such). `print $3` is the instruction to only print the third column.

`operator()(char)` writes the character to the pipe that is connected to the UNIX command's standard input. It can then see if data is available from the read-pipe that is connected to the command's standard output. However, data may not be available (yet). Therefore, we must use a non-blocking read for this. If there is no data available, we do nothing.

There is a minor complication. Some UNIX filters (such as `sort`) do not start outputting data until they have read all data up to end-of-file. Then they start processing and outputting data. However, our `operator()(char)` does not know when to create that end-of-file condition. (In UNIX an end-of-file is forced on a pipe by closing the write end of that pipe). Therefore we have to make one more virtual function in our filter family: `close()`. This function will close the write end of the pipe, read from the read-end of the other pipe until it itself gets end-of-file and pass all the data on⁵.

After it has read the pipe empty, it has to correctly wait for its child process (using the `wait()` system call) lest zombie processes will be created. Note that the `close()` must know where to leave the data. Therefore it is called by `ofiltstream::close()`.

Having implemented `unixfilter`, we can truly say we have brought UNIX filters to `iostream`.

To summarizing, a filter has to implement:

- `operator()(char, ostream&)`
- `clone()`
- a copy constructor if the default copy constructor is not good enough.
- a `close()` function if the filter has to be notified that buffered data must be flushed and the filter will no longer be used. Most filters do not need this. The notable exception being `unixfilter`. `close()` will be called by the `ofiltstream::close()` which is also called by `ofiltstream::~~ofiltstream()`.

Performance considerations

The `unixfilter` writes characters to the pipe using the `write()` system call. And after writing, it tests for available characters in the other pipe using a non-blocking read system call. Compared to simple operations, `write()` and `read()` are expensive. They cause a jump to kernel mode and may cause a context-switch - a non trivial operation. And, even worse, most of the times, there will be no characters

5. This is the reason we have not implemented `unixfilter` for the templated setup. Because `filtstreambuf<T>::close()` has to call `T::close()`, either every filter type would have to implement the `close()` member function, or we would have had to use some traits like `setup` to test at compile time if `T::close()` is available. Given the minor performance difference, we decided not to follow that road. Having a *virtual* function `filter::close()` that does nothing solves the same problem for the other two designs (strings and streams based filters).

available for `read()` to pick up (yet).

Therefore, `unixfilter` buffers characters. This helps in three ways:

- It causes fewer `write()` system calls.
- It causes fewer `read()` system calls, most of which are likely to fail anyway.
- The amount of data that is retrieved from a successful read is larger. Hence fewer write calls are needed within the `unixfilter` to the associated `ostream`.

The performance gain is quite large, as can be seen in the next section (section 9).

9. Performance

We have tested the performance of the `ofiltstream` class using the testbed below:

```
for (int i=0; i<nloops; ++i) {
    stream << 123456789 << ' ' << std::rand() <<
        " abcdefghijklmnopqrstuvwxyz" << '\n';
}
```

The `rand()` call is done to make sure not all lines are equal (good for the UNIX sorting and UNIX gzip tests). On average, the lines will be about 46 characters long.

In the code snippet above, `stream` is the `ofiltstream` under test. `ofiltstream` was either given `std::cout` or an `ofstream` as the output stream to send filtered data to. In both cases, the output is sent to a file on disk (in the case of `cout` this is done by using output redirection in UNIX' shell: `prog > outfile`

All testing was done on a Pentium IV 1.7GHz machine running RedHat linux 8.0; the compiler used was gcc version 3.2 20020903 (Red Hat Linux 8.0 3.2-7). The results were taken by running a test ten times, removing the fastest and the slowest run and then averaging out the times of the remaining eight runs. `nloops` was set to one million. Two types of tests were run: tests that do not use `unixfilter`, and those that do.

9.1 non-unixfilter tests

The non-`unixfilter` tests were:

`null`

Testing a very empty `ofiltstream`. This will output nothing. The data will not even flow through `cout`. Effectively, this measures the time needed to do all formatting and calling the `overflow()`, i.e. the mechanisms involved in `ostream`.

```
ofiltstream stream(nullfilt, std::cout);
```

`nofilt`

Testing the performance of the “normal” `cout/ofstream` case in the testbed. This adds the time needed for the physical I/O compared to the null test above.

```
std::ostream& stream=std::cout;
```

or, in case where an ostream is used:

```
std::ofstream out("out");
std::ostream& stream=out;
```

cat

This effectively measures the overhead of the ofiltstream in a very simple way: every character it gets, is immediately passed on to cout.

```
ofiltstream stream(cat, std::cout);
```

catcat

Testing the overhead of using a combifilter. Two cat filters are used to minimize any other overhead.

```
ofiltstream stream(cat | cat, std::cout);
```

catcatcat

Verifying the overhead of the use of combifilter. The outcome should approximately be the timing of cat with twice the overhead of catcat compared to cat (i.e. catcatcat = cat + 2 * (catcat - cat)).

```
ofiltstream stream(cat | cat | cat, std::cout);
```

rot13

Testing the overhead of encrypting⁶ data. An example of a simple filter that actually does something.

```
ofiltstream stream(rot13filt, std::cout);
```

reverse

Tests the reversing filter - another filter that does something to that data, but it is more elaborate than rot13.

```
ofiltstream stream(revfilt, out);
```

log

To test the overhead of adding time stamps. Note that this test also outputs more characters. 24 characters containing the time stamp are output before the original lines. Thus the "overhead" demonstrated in the nofilt test is approximately 50% larger.

```
ofiltstream stream(logfilt, std::cout);
```

tee

Tests making a duplicate of a data stream to a file.

```
std::ofstream split("split");
tee ttt(split);
ofiltstream stream(ttt, out);
```

teelog

To test the most useful application of our filters: having time stamped duplicates of the output stream sent to a file. Like catcat, this also uses a combifilter.

```
std::ofstream dbg("out.debug");
ofiltstream stream(tee(out) | logfilt, dbg);
```

6. Just kidding.

We have tested both outputting to `cout` and to an `ofstream`. This showed dramatic differences, as could be expected. Also dramatic were the tests where the cleaner string-based `ofiltstream` was used.

The numbers in table 1 are the times that the programs spent in user space. The system space timings were all similar, around 0.35 seconds. The main exceptions being `null` which had (not writing any data to the system) no system time clocked at all in any of the cases, `log` which spent around 0.90 seconds system time (due to the calls to `time()` and longer output lines), `tee`, which spent twice as much system time and `tee` plus the extra time `log` needed.

name	templated stream based		virtual stream based		virtual string based	
	cout	ofstream	cout	ofstream	cout	ofstream
null	5.22	5.37	6.90	6.99	16.36	16.44
nofilt	23.00	5.22	22.90	5.03	23.83	5.02
cat	34.45	17.58	34.91	19.02	43.80	26.89
catcat	47.38	27.76	50.37	31.17	86.43	70.10
catcatcat	57.92	35.05	66.57	44.62	130.79	113.66
rot13	37.84	20.80	38.32	22.25	47.61	29.52
reverse	39.46	23.23	40.60	24.96	43.65	25.29
log	50.96	22.15	51.28	23.89	62.67	33.32
tee	50.96	30.19	49.20	31.20	58.71	39.81
tee	61.91	45.84	70.45	50.63	107.76	88.22

Table 1. Timing data in seconds (user time) for the non-UNIX filters

Looking just at the `ofstream` results, we can clearly see that the templated version is faster than the two versions based on virtual functions. Especially when `combifilter` comes into play, the difference becomes significant. Yet, the notational inconvenience that comes into play when using `combifilter` is also major. For example, for the `catcatcat` case, the `ofiltstream` definition is:

```
ofiltstream<combifilter< combifilter<felix , felix > , felix > > stream(out);
```

Comparing the two virtual implementations, we see that if there are no `combifilters` involved, the difference in time used is more or less constant (except for the `reverse` test, but there even the streams based version has to do a lot of string related work). When `combifilters` are involved, the string based version is a lot slower due to the higher number of string construction, copy and destruction operations needed.

9.2 UNIX related tests

The timing data for the `unixfilters` show the same explainable difference between the `ofstream` and the `cout` variants, therefore we have not included the data for the `cout` variant. The UNIX related tests were:

```
unixcat
```

To test the performance of piping data to a UNIX command. Note that data is

written to the pipe, read from the other pipe and then written to `cout`. Hence (compared to the `cout` case above) three in stead of one I/O operations are needed.

```
ofiltstream stream(unixfilter("cat"), std::cout);
```

unixcatnull

Trying to eliminate the read from the pipe and the write to `cout`.

```
ofiltstream stream(unixfilter("cat > /dev/null"), std::cout);
```

unixsort

Uses a filter that will produce all output after the stream itself has been closed. The sorting key is the (random) number on the line.

```
ofiltstream stream(unixfilter("sort -k2n"), std::cout);
```

unixgzip

Testing a CPU-intensive filter: a compression program. If `nloops` in the program is set to one million, 47,482,435 characters are written to `gzip`, and 6,150,795 characters are read back.

```
ofiltstream stream(unixfilter("gzip -9"), std::cout);
```

unixwc

Making use of a UNIX filter that consumes a lot of data and produces very little data.

```
ofiltstream stream(unixfilter("wc"), std::cout);
```

name	elapsed	parent		child	
		user	system	user	system
unixcat	7.89	7.72	0.07	0.01	0.05
unixcatnull	7.70	7.57	0.05	0.01	0.03
unixgzip	17.95	7.64	0.10	10.04	0.05
unixsort	15.49	7.90	0.11	6.65	0.45
unixwc	8.05	7.52	0.05	0.40	0.02

Table 2. Timing data of the streams implementation.

name	elapsed	parent		child	
		user	system	user	system
unixcat	17.92	17.40	0.15	0.01	0.05
unixcatnull	18.02	17.46	0.11	0.02	0.03
unixgzip	28.45	17.56	0.17	10.06	0.07
unixsort	25.46	17.50	0.20	6.69	0.44
unixwc	18.39	17.46	0.10	0.42	0.03

Table 3. Timing data of the string implementation.

Here too, we see a constant difference in time between the streams based and the strings based versions. Because the strings are much larger (based on a single read from the pipe), construction, copy and destruction overhead is far less prominent, and thus less influenced by the amount of data coming back from the `unixfilter`

(compare `unixcat`, `unxsort` with `unixcatnull`, `unixgzip`, `unxsort`, `unixwc` which have far less data coming back from the UNIX filter).

9.2.1 Buffering in `unixfilter`

The buffering in `unixfilter` is very necessary. Let us, as an example, take the `unixcat` test. This took on average 7.89 seconds to run. In all, 47,482,435 bytes of data were transported to and from the UNIX `cat` program. As the buffer size was set to 4,096 bytes, this leads to some 3,000 context switches per second. If the buffer size is set to 1, thus disabling buffering completely, the running time goes to 430 seconds and some 220,000 context switches per second. Most of the time is spent in system mode. Making the buffer larger than 4,096 bytes will not help, as the pipe size in Linux is 4,096 bytes. A write to a pipe will be broken up in smaller writes if the write-size is larger than the size of the pipe.

10. Conclusions and wish list

Adding new streams to `iostream` in itself is not trivial. We have shown a way in which (at the cost of efficiency) writing new streams is easy. In this case, our model makes it easy to filter data that is written by an `ostream`. Though the model we have implemented is not complete, it is already very useful for debugging and filtering data. For broader use, improvements and changes are needed.

The implementation currently is not (yet) exception safe. Also, having seen how much the buffering in `unixfilter` saves, we should find some way not to turn buffering off in `ofiltstream`. This has to be done without interfering with the filters' work (for example, for accurate time stamps, the logging filter has to get the first character after a newline character at exactly the right moment, so line buffering is useless here).

Of course, having tasted the convenience of `ofiltstream`, we would also like to have an `ifiltstream`. If `ifiltstream` would allow a UNIX pipe as filter, any program reading from a file could make use of the services⁷ the C pre-processor provides. This would allow any program to support macros and defines in configuration files and such:

```
std::ifstream configfile("myprog.conf");
ifiltstream is(configfile, unixifilt("cpp"));
```

It would also be nice if filters could be inserted and removed from streams at will. Of course this is already possible by introducing the filter in a deeper nested scope. Manipulators could should be available to turn filters on and of, or even to configure the format the filter uses. This last feature would especially be useful for the logging filter. Which of course, should use locales to format the time stamp.

7. Another joke?

11. References

- The ISO/ANSI C++ Language Standard.
ISO/IEC International Standard 14882
International Standard for Information Systems - Programming Language C++
- On IOStreams
Angelika Langer and Klaus Kreft
Standard C++ IOStreams and Locales - Advanced Programmer's Guid and Reference
Addison Wesley, 2000
ISBN 0-201-18395-1