This is a pre-publication draft of the column I wrote for the November-December 1995 issue of the *C++ Report*. "Pre-publication" means this is what I sent to the *Report*, but it may not be exactly the same as what appeared in print, because the *Report* and I often make small changes after I submit the "final" draft for a column. Comments? Feel free to send me mail: `smeyers@aristeia.com`.

# The little endl that couldn't

Waste not, want not. It's true. Look for ways to use what looks like scrap, and you never know what will turn up.

When I'm developing a program, I turn it over to friends when the program has achieved a state that is something more than a prototype, but less than a beta release. "Tell me everything that's wrong with it," I'll ask, and, if I'm lucky, they'll oblige. It's not always great for my ego, but it certainly improves my programs.

These days I spend more time writing than programming, but I've come to discover that the two tasks have much in common. As a result, when I'm working on a book and I think I'm close to being finished, I turn it over to reviewers and ask them to tell me everything that's wrong with it. I then revise the manuscript, taking the reviewers' comments into account.

As I write this in August, I'm in the process of finishing work on *More Effective C++*. This column is due for publication around November, so the book may be available by the time you read this. (If it's not, it probably means my release schedule slipped. I *told* you the tasks were similar.) Last month I got back the reviewers' comments on my draft manuscript, and one of the changes I've made is the elimination of one Item I'd planned to include in the book. Even though the Item isn't suitable for *More Effective C++*, it's instructive to give it a look anyway. What follows is the Item that won't be, a *More Effective C++* outtake. After the Item, I'll explain why I decided the material isn't ready for prime time.

**Avoid gratuitous use of endl.**

Sometimes the most insignificant-looking things can make a big difference. For example, some people use the manipulator `endl` as a shorthand for inserting a newline into an output stream. That is, instead of writing this,

```
cout << "The value of x is " << x << '\n';
```

they write this:

```
cout << "The value of x is " << x << endl;
```

Perhaps the reason they do this is they think the two expressions are equivalent, and they use the `endl` version because it's easier to type. Well, it *is* easier to type, but `endl` is not equivalent to a newline, and the difference between the two

should be enough to convince anybody that there's more to writing programs than finding the easiest sequence of characters to type.

Inserting `endl` into an output stream does two things. First, it inserts a newline. That's hardly news. Second, it flushes the stream's buffer, thus forcing any pending output to be immediately written. From an efficiency point of view, this latter action is a cause for concern.

There's a reason why streams are buffered in the first place: writing a whole bunch of characters to an output device (or reading a whole bunch of characters from an input device) is, on a per-character basis, much less expensive than is writing (or reading) the same number of characters one by one. Furthermore, many I/O operations can proceed perfectly acceptably even if the actual physical operations are buffered. For example, most programs don't much care exactly when the data destined for an output file appears in the file as long as it's there by the time the program exits. Buffering, therefore, is a way for the I/O system to improve the efficiency of a program without changing its behavior. That's why the I/O subsystems of virtually all programming languages employ buffering.

Of course, sometimes you want to ensure that the data you insert into a stream is written right away. For example, error messages should typically not be buffered — they should appear immediately. Here's something you might find in an ANSI C compiler:

```
cerr << "Sorry, unlike K&R C, in ANSI C the"
     << "numeral '8' is not an octal digit.\n";
```

When you write to `cerr`, however, there's still no need to use `endl`, because the designers of the iostream library, realizing that error messages shouldn't be buffered, specified that `cerr` not buffer its output. So it doesn't.

If you have a stream object of your own that you'd like to go unbuffered, you can set it up so there's no need to manually flush the buffer each time you write to the stream. Configuring such a stream is a little tricky, however. You do it via the following far-from-intuitive machinations, in which I assume you wish to set up an unbuffered `ofstream` object, `errorFile`, that corresponds to the output file `errors.txt`:

```
// create an ofstream not associated with any file
ofstream errorFile;

// disable buffering for the ofstream
errorFile.setbuf(0,0);

// associate the ofstream with "errors.txt"
errorFile.open("errors.txt");
```

From here on out, you can use `errorFile` just like any other `ofstream` object, and you can revel unreservedly in its unbuffered glory.

As we've seen, you don't need to use `endl` with `cerr`, but perhaps you need it with `cout`. After all, a read from `cin` is typically preceded by a prompt written to `cout`, and it would

be quite the pity if the read operation were attempted before the prompt had been displayed, wouldn't it? Not to worry. Those infamous iostream designers have beaten you to the punch once again. `cin` and `cout` are tied together such that any attempt to read from `cin` will automatically cause buffered output destined for `cout` to be flushed. Prompts written to `cout` will therefore always appear before reads from `cin` are initiated.

Even if `cin` and `cout` didn't coordinate their activities, using `endl` to terminate a prompt would produce ugly output, because the newline component of `endl` would force the user to type the desired input on the line *after* the prompt. It's usually much more professional-looking to have them type the requested input on the *same line* as the prompt — like this:

```
string name;

cout << "Please enter your name: ";
cin >> name;
```

In this example, it's unfortunate that `name` has to be constructed before it can read the value it's supposed to initially hold, but we're required to have an existing `String` object before we can call `operator>>`, so we're stuck with this small inefficiency. *C'est la C++*.

You, too, can tie an input stream to an output stream, thus guaranteeing that the output stream is flushed before input is read. You do it using a function called `tie`. You invoke `tie` on an input stream and pass it a pointer to the output stream to which you want to tie it:

```
ostream promptStream;
istream inputStream;

// always flush promptStream before reading from
// inputStream
inputStream.tie(&promptStream);
```

Okay, you don't need `endl` to flush messages to error streams, you don't need it for prompts, and you don't want to use it as a simple shorthand for a newline. In fact, the only time you *do* want to use it is if you're writing to a buffered stream and you really, truly want to both insert a newline and force an immediate flushing of the stream's buffer. Such times are rare, and your uses of `endl` should be equally rare.

And yet, it really *is* a pain to have to type `'\n'` each time you want to write a newline. If `endl`'s out, what's in? One thing you might consider is a little judicious pollution of your global namespace. You might define a suitable constant character as follows:

```
const char NL = '\n';
```

Now you can enter a newline into an output stream as easily as you like, but you won't have to cripple your program's carefully crafted I/O buffering mechanism to do it:

```
cout << "The value of x is " << x << NL;
```

Hey! This is even *shorter* than `endl` — only two characters! And if the pollution of the global scope offends you (it should),

you can even place your newline abbreviation in a personal namespace that you only import when needed:

```
namespace PersonalStuff {
  const char NL = '\n';
  ...
};
void printValue(int x)
{
  using PersonalStuff::NL;
  cout << "The value of x is " << x << NL;
}
```

As you can see, there's nothing wrong with shorthands, but it is important to make sure that the shorthands you use mean what you want them to mean. `endl` usually doesn't, so you shouldn't use it unless you're sure you really want both a newline *and* a flush.

* * * * *

Okay, that's the Item as I'd planned to have it in the book. Why did I decide to exclude it?

This particular Item was part of a chapter on efficiency, and some reviewers thought the efficiency gain by following the advice above wasn't worth the amount of text I devoted to the topic. That's probably true, but unnecessary use of `endl` happens to be one of my pet peeves, so I probably would have kept the Item anyway. After all, it's my book, and if you can't say what you want in your own book, when can you?

One reviewer (Eric Nagler) brought up a more important problem. He wrote:

> The user is going to have the buffer flushed, like it or not, because of the fact that the `ios::unitbuf` bit is turned on by default. So using `'\n'` vs. `endl` is a moot point. If the buffer really should not be flushed, then this bit needs to be turned off.

Had I been living under a rock? I'd never heard of `ios::unitbuf`. So I turned to Steve Teale's book on IOstreams [cite it here]. There I found that if `unitbuf` is set, an `ostream`'s buffer is automatically flushed after each write operation, just as Eric had written. (This is illustrative of why I am firmly committed to having reviewers.) However, Teale's book also said that the default setting for `unitbuf` is implementation-dependent, so I was hoping I could still squeak the `endl` Item in. After all, if many implementations defaulted `unitbuf` to 0, my advice would still be valid much of the time, and Teale's book implied that it was common to default `unitbuf` to 0.

Just to be on the safe side, I looked `unitbuf` up in the April 1995 draft ANSI/ISO C++ standard document [cite it here]. I was unable to determine the default setting for the bit, but while wandering around the `iostream` portion of the standard, I noticed something that took me by surprise: `cout` is supposed to be *unbuffered*. You

may not find this to be a great shock, but traditionally `cout` has been a buffered `ostream`. For unbuffered output, the traditional stream of choice has been `cerr`. It seems the ANSI/ISO committee decided to part with tradition.

My guess is that `cout` is still buffered under many (if not most) implementations, but the standard will be the standard, so in the future, `cout` will not be buffered. Yet most of the places where I observe `endl` being used gratuitously, the destination stream is `cout`. Under those conditions, the call to `endl` is still unnecessary (the stream is already being flushed after each write), but it's no longer defeating the buffering system as I said it was in my would-be Item above. Once implementations come into conformance with the standard, then, my explanation would have been misleading.

I hope my book will be useful long after implementations have started to conform to the standard, so I decided to abandon the Item now instead of regretting its inclusion later. As long-term advice, it just couldn't cut the mustard. For a shorter-term publication like a magazine column, however, it suffices just fine.

Waste not, want not :-)

**References**

Steve Teale, *C++ IOStreams Handbook*, Addison-Wesley, 1993.

Accredited Standards Committee X3J16, "Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++," April 1995.