Type-Safe Formatting

Andrei Alexandrescu

November 29, 2005

Programming in the Windows environment has become a great deal better lately—more precisely, ever since a lot of the Unix tools have been ported to it. I've programmed for a long time under Windows and then switched to Unix. After having the quintessential "Unix ripped my leg and beat me to death with it" experience, followed by the "Hey, I never thought things can be done *that* cool way!" experience, I can say that programming under Unix can be a lot of productive fun. The usual invariants apply; a programmer's "goodness" will port across platforms, but for a given programmer, Unix might be better after getting over its learning curve (unless, of course, they put you in a hospice first). Also, I'm not going to say I'm a whole different person, now that I know the "Don't know how to make love" joke. But one thing is true—of my programmer friends, the ones who love Windows are exactly those who don't know Unix. (Mysterious face and tone.) Coincidence?

In fairness, there's been a lot of cross-pollination; competing with Windows has made Unix (in particular Linux) a lot easier to use for novices, and maybe more comfortable even for experts. The windowing environment (KDE) that I'm taking for granted under Linux is, I could say, even more sleek that Windows in places, and definitely more configurable. But for all the Explorer-like file manager and other graphical tools, you'll always see a command line window on my desktop. Because I kid you not, a good command is worth a thousand mouse clicks.

Much ink has flown, and many trees have been killed, to bear writings comparing of the two (and other) platforms, by people much more knowledgeable and more talented than yours truly (check for example the classic "In the Beginning was the Command Line" [4], a very enjoyable read), so I won't go about comparisons any further. But there is one thing—and this segues nicely into this article's theme—that makes the Unix lifestyle attractive to people passionate about programming languages: any programming task you want to get done, there's not one, but a full toolchest of one-syllable little programs (awk, perl, sed, grep, bash. . . —they sure must name bodily parts in Klingon) that are eager to help. Much everything is a living language in Unixland, something that makes me think of the Windows environment as rigid, constrained, and lacking freedom of expression.

So, having to do some simple file processing tasks under Unix, I naturally used the wonderfully messy Perl, and that worked great up to a point where I needed more speed. So I decided to rewrite that simple utility in C++. And believe me, if you want to miss Perl, the perfect approach is to use C++'s I/O formatting amenities.

1 `cout << "Why oh why, world?";`

Let me say it upfront and in all honesty: I'm not a fan of C++'s iostreams. They are cute at first sight, but in many ways I don't consider them a big step forward compared to C's `printf`. Given that I hold `printf` in contempt, you can easily imagine the sorry reality of my existence.

The old `printf` family of functions has the advantages of speed, conciseness, and separation of format from data, at the cost of safety. Indeed, `printf` is about as well-intended, fast, and prone to disaster as a playful Great Dane running in a porcelain boutique. So then came along C++'s iostreams, which are much more safe, but totally forgot the lesson of separating format from data. To clarify what that means, imagine formatting some message. With `printf`, you'd go about things something like:

```
printf(
  "Heh, %u frobs and %u twids\n",
  frobs, twids);
```

An important point is that the format string is separate from the data, so it's easy to put it in some file, give it a symbolic name, and then write:

```
printf(FMT_FROBTWIDREPORT,
  frobs, twids);
```

The file defining `FMT_FROBTWIDREPORT` collects all formatting messages and might be under the control of a different team that deals with the user experience, translations, and so on. (There's one problem lurking in the back—what if some foreign language wants to render the parameters in a different order? We'll get back to the issue of positional parameters later.)

With iostreams, things look like this:

```
cout <<
  "Heh, " << frobs <<
  " frobs and " <<
  twids << " twids\n";
```

Safety is back, at the cost of a syntax that looked pretty cool in the 80s (but then hey, you'd be amazed at some of the music that was cool in the 80s). Anyway, no more need to carefully pair the `"%"` directives with the count and the types of the passed-in arguments—operator overloading takes care of that. But out of the window is the useful separation of data from format. And if you enjoy clunky syntax and format from that's inseparable from data, you'll really love to hear that iostreams are sluggishly slow as well. To just print the line above to a file on my system (Linux, of course), `printf` takes on average 1.78 milliseconds, while `cout` takes 2.67 milliseconds—that's a whole 50% bottom line slowdown due to formatting costs alone! And we're not talking about a dry run—that's writing real bits to a real file on a real disk. No matter how many times I've heard that iostreams can be implemented efficiently, and no matter how I/O bound a program that writes a million lines to a disk is, the reality in the field is that iostreams are surprisingly good at slowing things down. In Oliver Schoenborn's own tests [3], iostreams fare as bad as 250% slower than `printf`.

But wait, there's more. Oh, and a lot more—more code you have to write to format things. A particularly eloquent (and not even extreme) example can be found at `http://noptrlib.sourceforge.net/utils/coutf/`, which actually describes Oliver Schoenborn's `coutf` library [3]. Basically, to achieve the effects of:

```
printf(
  "Hi, count=%s, time=%s, radius=%-6.2fs",
  count, theTime, radius, eol);
```

you'd have to write the following iostreams–based code:

```
const int savePrec = cout.precision();
const int saveWidth = cout.width();
const fmtflags saveFlags = cout.flags();
// output:
cout << "Hi, count=" << count
  << ", time=" << theTime << ", radius="
  << ios_base::left << setw(6)
  << setprecision(2)
  << ios_base::fixed << radius
  << std::endl;
// restore formatting state:
cout << setprecision(savePrec)
  << setwidth(saveWidth);
cout.flags(saveFlags);
```

If iostreams are a step towards the future, I sure hope the future will have a definitive solution for the Carpal Tunnel Syndrome.

Googling for "safe printf" yields (in addition to the `coutf` library) a number of results, among which the Format library (part of Boost) [1]. With Boost Format, you'd write:

```
cout << boost::format(
  "Heh, %1% frobs and %2% twids\n")
  % frobs % twids;
```

The Format library is typesafe, supports positional parameters (by numbering them; the same argument can be printed several times by repeating its number in the format string), and provides pretty good compatibility with `printf`. However, I decided Boost Format wasn't for me, mainly because it lacks a typesafe `scanf` counterpart, which I needed as well. Also, Boost Format and `coutf` both build on top of iostreams, which I didn't like to begin with, and which will guarantee slow execution. Why work hard to implement the good fast simple behavior on top of the new bad slow infrastructure, instead of throwing everything away and enjoying the pleasure of reinventing the wheel all over again?

Some little nits: the `coutf` library only supports up to nine parameters, which is "cheating." With Boost Format, I wasn't that thrilled about overloading "%" for passing arguments, but that has to do with my having settled long ago on a different solution for variable arguments.

2 The Trailing Parens Idiom

When it comes about functions with variable argument count, it's safe (in more than one sense of the word) to flat out unrecommend the built-in ellipsis facility. The reasons are discussed in a number of places. Suffice to say that in a little (40K lines) application that I inherited, so far I found a total of four bugs: one was an `assert` with side effects, and the rest were varargs-related. Ironically, they were supposed to format error messages, the very point at which you'd love to see some output before the application dies on you!

A simple way to emulate variable arguments is to have your function return an object that accumulates state and has a member function that adds more arguments, and returns a reference to `*this` (such that calls can be chained). This allows the client syntax:

```
Function(expr1).add(expr2).add(expr3);
```

You only need to choose an appropriate name for the `add` function... or should it be `with`, `pass`, or even `_`? You could overload some operator (that's what Boost Format does), yielding the client syntax:

```
Function @ expr1 @ expr2 @ expr3;
// or
Function(expr1) @ expr2 @ expr3;
```

where `@` is your operator of choice. The only trouble is that now your chosen operator will be in the precedence game with $\texttt{expr}_n$ that you pass, so you need to be careful.

To avoid both the dilemma of choosing a name for the trailing function, and the dilemma of finding an operator that looks good and has the right precedence, I've settled for the most concise solution of all, which is simply overloading `T& T::operator()(U)`, where `U` ranges over the types of interest. That fosters a very simple client syntax:

```
Function(expr1)(expr2)(expr3);
```

The call syntax is simple and uniform, and the first parameter doesn't have special syntactic status compared to the others.

3 Type-safe Formatting

So by now it's no secret what this article is up to: Define a function that is format–string–compatible with `printf`, uses the trailing parens idiom to collect arguments, and of course is type-safe as a natural outcome of using overloading instead of the dangerous ellipsis. Nothing new here, but so very useful. In Petru Marginean's words, "when starting on a job, I always bring `ScopeGuard`, `Enforce`, and typesafe `printf` in my little virtual backpack."

Let's crank up our ambitions a bit. The `printf` family also includes `fprintf` and `sprintf` (plus its cousin `snprintf`, which is supposed to be a teeny bit safer). So let's define not only functions that can write to files and strings, but functions that can write to any device supporting a character I/O operation. To keep it simple, the smallest interface that makes sense is a function:

```
// Implement this for your Device type
// and your Char type
void write(Device where,
  const Char* begin, const Char* end);
```

(Don't get fooled by the pass–by–value of `Device`; `Device` is a generic type that actually could be a reference or pointer type.) This design starts so sweet and simple, it's hard to contain the impetus of implementing some useful devices right on the spot:

```
void write(std::FILE* f,
    const char* from, const char* to) {
  assert(from <= to);
  // TODO: throw on error
  fwrite(from, 1, to - from, f);
}
void write(std::string& s,
    const char* from, const char* to) {
  assert(from <= to);
  s.append(from, to);
}
```

Next thing we need to do is to define a class, let's call it `PrintfState`, that holds the formatting string and implements overloads of `operator()` to print to a device. `PrintfState` is templated on the `Device` it works on, and also on the character type it uses:

```
template <class Device, class Char>
class PrintfState {
public:
  PrintfState(Device dev,
      const Char* format) {
    ... initialize state ...
  }
  ... more ...
};
```

We'll get back to the implementation shortly; first, let's complete the scaffolding by defining the user-invocable functions:

```
PrintfState<std::FILE*, char>
Printf(const char* format) {
  return PrintfState<std::FILE*, char>(
    stdout, format);
}

PrintfState<std::FILE*, char>
FPrintf(FILE* f, const char* format) {
  return PrintfState<std::FILE*, char>(
    f, format);
}

PrintfState<std::string&, char>
SPrintf(std::string& s, const char* format) {
  return PrintfState<std::string&, char>(
    s, format);
}

template <class T, class Char>
PrintfState<T&, Char>
XPrintf(T& device, const Char* format) {
  return PrintfState<T&, Char>(
    device, format);
}
```

Of course, you could create instances of `PrintfState` by hand, but the functions have the advantage that they deduce their template arguments, so there's no need to specify them. The first three functions define counterparts for `printf`, `fprintf`, and `sprintf`, respectively. The last function, aptly named `XPrintf` (for some reason, "X" holds the title of the coolest letter in the alphabet… "X" sells!), defines a generic wrapper for whatever new device you have invented. By the way, if you want to define output to something as fast as a straight fixed-size character buffer, all you have to say is:

```
template <class Char>
void write(pair<Char*, Char*>& s,
    const Char* b, const Char* e) {
  assert(b <= e);
  if (e - b > s.second - s.first)
    throw overflow("bad luck");
  s.first = copy(from, to, s.first);
}

template <class Char, size_t N>
PrintfState<pair<Char*, Char*>, Char>
BufPrintf(Char (&buf)[N], const Char* format) {
  return PrintfState<pair<Char*, Char*>, Char>(
    make_pair(buf, buf + N), format);
}
```

The code above simply improvises a device out of two pointers marking the buffer's boundaries (very STLesque, I know). Each call to `write` will do the obligatory (please!) bounds check (which is obligatory), then copies the data and bumps the pointer (oh, did I mention the obligatory check preceding all that). If you prefer, you could use a smarter fixed-size vector (Boost has one) as a back-end, while, of course, not forgetting to perform the obligatory bounds check.

Let's get back to `PrintfState`'s implementation. The rules that govern `printf` argument-taking ways (and, in fact, those that govern ellipses) are:

1. All integral types are converted to **`long`** (or **`unsigned long`**, doesn't matter, since signed and unsigned types have the same representation);

2. **`float`** is converted to **`double`**;

3. Pointers to objects become **`void*`**;

4. Everything else is undefined.

We'd like to emulate that behavior, except for (4) of course. So `PrintfState` defines overloads of **`operator()(unsigned long)`**, which does the actual work, and then defines **`operator()`** for all integral types to forward to **`operator()(unsigned long)`**. You can do that by hand, with the help of a template, with type traits, or (if you're lazy like me) with a lowly macro. (Just give the macro a really really long name.) Then, `PrintfState` defines **`operator()(const char*)`**, **`operator()(int*)`**, **`operator()(short*)`**, and **`operator()(long*)`**, the last three for implementing the `"%n"` specifier that allows one to save how many characters have been printed so far. **`operator()(const void*)`** defines the implementation-dependent rendering of pointers (via the `"%p"` specifier). Finally, `PrintfState` defines **`operator int() const`** such that, in the name of `printf` compatibility, callers can fetch the total number of characters written (`-1` in case of error).

The strategy employed by `PrintfState` is necessarily different from that of `printf`, due to the incremental way that parmeters are revealed to `PrintfState`. Upon initialization, `PrintfState` makes as much progress as it can (by printing its format string up to the first directive). Then, as **`operator()`** is invoked, `PrintfState` formats one argument, writes it out, and then again writes the format string up to the next formatting directive.

4 Testing 1–2–3

Cranky old man that I am, I not only dislike the iostreams and `printf`, but also testing my code. In fact, my entire career path could be debunked as a series of moves to keep me away from having to test code. First, I strived to be a manager; then, I've tried consulting; then, I cunningly wrote Loki such that no compiler could even compile it, so there was nothing to test; and finally, I've become a grad student. You can't dodge testing much better than that.

Yet, after attending one of Robert Martin's hilarious tutorials (highly recommended) at a conference, something stuck to my ear: "I'd gotten used to always first write a test that fails, and only then write code to make the test work." I've tried this very simple technique and I have to say it's highly satisfactory. Implementing features without thorough regression testing leads to an increasing feeling of insecurity. On the contrary, when you first write a test, and then write code to fulfill it, gives a pleasant sense of utility (hey, your test code already is the first satisfied customer).

In the case of `Printf` testing actually is easy, because a reference is readily available: `printf` itself. So a test case would format some stuff using `snprintf`, then format the same stuff with the same format string, and compare the results.

Concocting format strings that exhaustively test all of the combinations of various specifiers and modifiers turned out to be a boring task, so why not leave it to chance—and quite literally so. My test code generates some random letters, followed by a random (but correctly formed) format specifier, followed by some more random chaff. Depending on the type character generated, a different type of randomly-generated data (integral versus string versus pointer) is passed to the comparison routine. Rinse, lather, repeat in an infinite loop. At the time of this writing, the code has passed 32,623,234 tests. Wait, that's 33,236,174.

Seeding the random number generator with a constant number yields a pseudorandom sequence that repeats exactly the same every run. That eases debugging; if your program crashes at iteration 6455, inserting a conditional breakpoint reproduces the problem exactly. If exact reproducibility is not needed, seeding the random number generator with `time(0)` will yield a different sequence each run.

5 More Features

So far `Printf` offers no more formatting amenities than `printf`, plus some more device independence. You can download the source code from `http://moderncppdesign.com/code`. Once this baseline is in place, many extensions come to mind:

- *Positional parameters.* If you care about ever translating your strings to a different language, positional parameters are a must.
- *More formatting directives.* Printing out arrays, strings, STL containers, and the such are immediate. A more general extension mechanism that would allow users to define their own formatting directives is just around the corner. (In fact, GNU's `printf` implements the C version of exactly that idea as an extension, see `http://pauillac.inria.fr/~lang/hotlist/free/licence/fsf96/drepper/paper-7.html`.)
- *Scanf.* No matter what amenities `Printf` implements, their respective converse should be implemented by `Scanf`.
- *Regular expressions.* Now that we got to talk about `scanf` and Perl, how about a input scanning library that supports regular expressions? I'd love that, and I guess I'm not the only one. Eric Niebler's xpressive library [2] is an xcellent implementation or perl regexes, and is asymptotically approaching version 1.0. Adding formated input primitives on top of that should be "a trivial matter of programming."

I got so excited about "printfing" stuff, that I completely forgot about the Mailcontainer section, and now there's no space anymore. (You see, CUJ has a fixed buffer design for their number of pages.) That is a pity since many gentle readers have sent a lot of great ideas and feedback following the article about generic Observers, including the ego–shattering "I am kind of unsatisfied with the hierarchal policy idea that you proposed in CUJ." Please bear with me until the next installment of Generic⟨Programming⟩. Til then, let me restore my ego by eyeing `Printf`'s test count: wow, 88,223,546 passed.

References

[1] Samuel Krempp. The Boost Format Library. Available at `http://boost.org/libs/format/doc/format.html`.

[2] Eric Niebler. The xpressive Library. Available at `http://boost-sandbox.sourceforge.net/libs/xpressive/doc/html/index.html`.

[3] Oliver Schoenborn. The `coutf` Library. Available at `http://noptrlib.sourceforge.net/utils/coutf/`.

[4] Neal Stephenson. In the Beginning was the Command Line. Available at `http://cryptonomicon.com/beginning.html`, 1999.