Overloaded conversion as an alternative to printf

Greg Nelson
Imaging Systems Laboratory
HP Laboratories Palo Alto
HPL-2004-163
September 28, 2004*

type-safety, programming language output primitives

This note proposes a programming language feature for formatted output that is almost as convenient as printf but provides significantly better static and dynamic checking.

* Internal Accession Date Only
Approved for External Publication

Overloaded Conversion as an Alternative to printf

Greg Nelson

September 29, 2004

Abstract

This note proposes a programming language feature for formatted output that is almost as convenient as `printf` but provides significantly better static and dynamic checking.

Since the Fortran feature FORMAT, most if not all programming languages have provided language features or library routines for formatted output of numeric data. Perhaps the most popular of the many approaches has been the C library function `printf`. `printf` takes a variable number of parameters. The first parameter is a string (the *format string*), the remaining arguments (the *output arguments*) are the values to be printed. They can be of various types. In this note, I will call the types that are allowed for the output arguments the *formattable* types. The format string contains within it several *conversion codes*, one for each output argument. The effect of `printf` is to send to the output a copy of the format string, with each conversion code replaced by the printed value of the corresponding output argument. For example, the conversion code "`%d`" specifies the conversion of an integer to a sequence of decimal digits. The conversion code may include type-specific formatting specifications. For example, the code "`%3.1f`" specifies the conversion of a floating point number to a three-character-wide decimal field with one digit to the right of the decimal point. Thus

```
printf("The floor of %3.1f is %d\n", 25.0/10.0, 2)
```

will print

```
The floor of 2.5 is 2
```

followed by a newline.

The varying number and varying types of the arguments are keys to the convenience of `printf`, but these features make it difficult to provide good static or dynamic checking for `printf`.

This little paper proposes a formatted output feature that is almost as convenient as `printf`, but provides better checking. It is not my suggestion to change C or C++ or their libraries. But, as new languages continue to be designed, and language design continues to be of scientific and engineering interest, the feature I describe may be useful in some designs in the future, even if it is moot for the great designs of the past. The reader should imagine a clean slate.

I suggest one left-associative overloaded binary infix operator, say "&", which I propose to call *conversion*. There is one overloading of the conversion operator for each formattable type T. Let us use *Convert.T* as the name of the overloading of & for type T. *Convert.T* has the type $String \times T \rightarrow String$ and the following semantics: *Convert.T*(s, x) produces the string obtained by replacing the left-most conversion code within *s* by the value *x* formatted according to that conversion code, or produces an error if that code doesn't exist or is inappropriate for type T. For example,

```
    print("the floor of %3.1f is %d\n" & 25.0/10.0 & 2)
≡   print("the floor of 2.5 is %d\n" & 2)
≡   print("the floor of 2.5 is 2\n")
```

Comparing the example with overloaded conversion to the example with `printf`, we see that, using &, we can write a command that is semantically equivalent to the `printf` version. And the version with & is not significantly more difficult to type or (more importantly) to read. (Only a curmudgeon would complain that the ampersand key is less conveniently placed than the comma key.)

Several aspects are interacting in this design. First, an overloaded left-associative infix operator is being used to avoid the need for a routine that takes a varying number of parameters of varying types. This aspect of the design is not new. For example, the C++ operator << uses the same technique. But the detailed syntax and semantics of & have been crafted so that there is an objective basis to the claim that & is almost as convenient as `printf` (objective by the standards of programming language design discussions), namely: the syntax and semantics of & are such that

> almost all occurrences of `printf` can be eliminated in favor of a semantically equivalent statement using & by simply replacing "`printf`" with "`print`" and replacing the comma before each output argument with an ampersand.

This strong argument for near-equivalence of convenience of & with `printf` is new. For example, the C++ operator << doesn't allow a format string or conversion codes.

Overloaded conversion provides better static and dynamic checking than `printf`. In the case that an output argument is not of a formattable type, or in the case that an output argument's type is inappropriate to the corresponding conversion code, the C language allows `printf` to print garbage without warning, which is just what the majority of implementations will do. In contrast, overloaded conversion will give a static type error in the first case and a runtime error in the second.

The naive use of overloaded conversion instead of `printf` will result in more string allocations and more string scanning, which are undesirable from a performance point of view. But optimizing away these unnecessary costs in the common cases would be easy.

I have read specifications for `printf` that define its behavior when the number of output arguments exceeds the number of format codes in the format string. The overloaded conversion operator that I propose will not duplicate these semantics. But the specifications that I have read differ, and I have never met a programmer who admitted to deliberately writing a program that relies on the specified behavior in this

case. So the fact that overloaded conversion doesn't duplicate these semantics cannot be claimed as a significant lack of convenience compared to `printf`.

It is time to explain the qualifier "almost all" in the claim providing an objective basis for the near-equivalence of convenience of & and `printf`. The necessity for this qualification arises from two circumstances.

First, for `printf`, *String* itself is a formattable type (with conversion code `%s`). In the uncommon case that an output argument of type string contains format codes, it is easy to come up with an example in which the mechanical transformation of printf into & fails. Thus, for example,

```
printf("%s %d", "%d", 6)
```

prints "`%d 6`", but mechanically replacing `printf` with `print` and commas with ampersands produces a different result:

```
print("%s %d" & "%d" & 6)
```

prints `"6 %d"` But it is for `%d` and `%f` that we love `printf`, not for `%s`. In a language that uses overloaded conversion as an output primitive, it would probably be best to avoid this problem by omitting *String* from the formattable types.

Second, `printf` will translate an occurrence of "`%%`" in the format string into a literal occurrence of a single "`%`" in its output, functionality that & cannot duplicate with equal convenience (since in general it takes more than one application of & to replace a single application of `printf`). If the "`%`" comes after the conversion codes in the format string, there is no problem. Otherwise, a workaround is to print an expression of the form A + `"%"` + B where + denotes string concatenation and A and B are applications of one or more occurrences of &. This is clearly less convenient than `printf`, but it is also not the common case.

A final subject must be touched on. Nowadays all programmers are expected to write programs that can easily be customized for use by people of different native tongues (three steps forward); and some programmers are even subject to the extreme requirement that the customization must consist only in substitutions for the format strings of the occurrences of `printf` in the program (two steps back). (The definition of `printf` has been changed to make the extreme requirement tenable.) If you are designing a programming language that you intend to be used by programmers who are subject to this requirement, you should not attempt to replace `printf` with overloaded conversion. But in this case you should stop designing a new language and just use C.

I am grateful to Mike Burrows, Rajeev Joshi, and Mark Lillibridge for their helpful comments on this paper.