Using Producer and Consumer Manipulators to Extend Stream I/O Formatting in C++

Michael Rizzo
Computing Laboratory, University of Kent
Canterbury, KENT CT2 7NF, UK
Email: M.Rizzo@ukc.ac.uk

Published in ACM SIGPLAN Notices 29(3), March 1994.

Introduction

The C++ iostream package makes use of the notion of stream manipulators, principally as a means of manipulating formatting state associated with a stream. This paper illustrates how parameterized manipulators which produce output and consume input can be defined to extend stream I/O formatting. Such manipulators can be especially useful for simple parsing of stream input.

Format state

There are two problems with stream format state:

- handling new formats can be messy, and
- it is all too easy to 'forget' the current format state when inputting values.

For illustration purposes, consider a manipulator `base(b)` which causes input and output of `int`s to be done in the indicated base.

To implement this manipulator in the typical format-state manipulator style e.g. as for the `hex` manipulator, a variable to hold the base must be added to the stream state. The `ios` class allows this to be done dynamically via the `xalloc()` member but the insertion and extraction operators for `int`s do not know about the base and consequently have to be re-defined. To make matters worse there is a feature-interaction problem as the following code segment demonstrates:

```
int i;
cout << base(3) << i << hex << i;
```

Clearly the implementation of the `hex` manipulator will also have to be changed for the above to be executed correctly. The same goes for `dec`, `oct` and `resetiosflags`.

The second problem is that when processing input it very easy to forget to 'turn off' a formatting manipulator. The effects of this may not become apparent until much later in the program's execution and it is not always immediately obvious as to what the cause of the problem is. For example, one may turn on the `hex` manipulator, and forget to turn it off when it is no longer needed. Later in the execution an input sequence of digits intended to be interpreted as a decimal integer will be interpreted as a hexadecimal one and the effects of this misread value may not be detected until much later, if ever[1]. Detection of such an error is even more difficult if it is execution-path dependent.

Producer and Consumer Manipulators

A *producer manipulator* is one which generates output on an output stream e.g. `endl`. Similarly a *consumer manipulator* consumes input from an input stream e.g. `ws`. We could define the `base` manipulator described earlier as a producer-consumer manipulator which does not require any stream state and which is only applied to its argument so that it does not need to be 'turned off'. For example:

```
int i, j;
cin >> base(12,i) >> base(4,j);
```

This treats the first value as a number in base 12 and the second as a number in base 4. The stream format state remains unchanged throughout. Implementation is easy and does not involve re-definition of any existing operations. Additionally each formatted value is clearly and explicitly associated with a format specifier, which is not always the case with format state manipulators.

There is one potential disadvantage with this approach which may be significant in some circumstances. Consider a vector class containing an array of `int`s. An insertion operator for such a class might output each of the `int`s in turn using the insertion operator for `int`. The format state approach enables all manipulators defined for `int`s to be used with the vector class. With the producer/consumer manipulator approach, new manipulators for vectors would have to be defined[2].

However there are several applications where this disadvantage is not an issue. In particular, the author has found the consumer manipulator technique to be especially useful when applied to string input as demonstrated in the following sections.

String Consumer Manipulators

A common source of irritation when reading a string from an input stream is that white space is used as delimiter. One well-known technique for specifying strings containing white space involves enclosing the string in quotation marks (as is the case with command line arguments in UNIX shells for example). Assuming the existence of a `String` class, we would ideally like to be able to write something like

```
istream is;
String s;
...
is >> s;
```

[1] This problem is no different from that of side-effects in programming languages like C, where evaluation of an expression may alter the system state.

[2] One could argue that this is actually safer, even if a bit long-winded.

so that if the incoming sequence of characters is `Hello there` then `s` is assigned the value `Hello` but if the incoming sequence is `"Hello there"` then the value assigned is `Hello there`.

The above code would force the extraction operator for class `String` to take care of processing quotation characters in this way. However this is not a very good solution because there are other ways of specifying strings containing white space and it wouldn't be wise to restrict our `String` class to a fixed set of possible representations.

The correct approach is to define a consumer manipulator to do the job so that this code becomes

```
istream is;
String s;
...
is >> quoted(s);
```

where the `quoted` manipulator takes care of processing quotation marks and assigning the value to `s`. This approach does not have any implications on the implementation of either of the stream or `String` classes. Different manipulators can be written to process other representations of strings containing white space.

Implementation is pretty straightforward: the standard IMANIP template for input manipulators can be used as described in Stroustrup pg 347, the only difference being that the argument is actually a *reference*[3]. Figure 1 contains the implementation of the `quoted` manipulator for the `String` class[4]. Note that this correctly handles error conditions, in spite of the apparent lack of error handling code.

Using String Consumer Manipulators to provide `scanf()`-like functionality

The `scanf()` family of functions may not be type-safe, but is richer than standard input streams in terms of its input processing capabilities[5]. Consider the example:

```
char buffer[BUF_LEN];
...
scanf("%*d BEGIN %[A-Z]", buffer);
```

This code reads and throws away an integer value, reads and throws away the characters 'BEGIN' and then reads a character string into `buffer`. This example demonstrates two classes of `scanf()`'s input processing capabilities, namely:

- character-eating (e.g. assignment suppression, format string matching),
- string-processing (e.g. matching incoming characters against a scanset).

Type-safety aside, `scanf()` is still far from perfect. It does not, for example, allow input to be matched against regular expressions. Worse still, `scanf()` is not extensible; adding features would require modification to its implementation.

[3] Manipulators taking more than one argument can be implemented in a similar fashion except that no standard template is provided.

[4] Using GNU g++ 2.4.5 with libg++ 2.4.

[5] This has led to some implementations of `istream` defining a `scan` function which works like `scanf()`. This approach defeats the whole purpose of streams because it suffers from precisely the same problems that `scanf()` does!

```
istream& quoted_str(
    istream &is, String &s
) {
  char c;
  is >> c;
  if (c == '\"') { // Quoted string
    s = "";
    bool end = FALSE;
    do {
      if (is.get(c)) {
        switch(c) {
        case '"':
          end = TRUE;
          break;
        case '\\':
          if (is.get(c)) s += c;
          break;
        default:
          s += c;
          break;
        }
      }
      else {
        end = TRUE;
      }
    } while (!end);
  }
  else { // String not quotes
    is.putback(c);
    is >> s;
  }
}

imanip<String&> quoted(String &s) {
  return imanip<String&>(&quoted_str, s);
}
```

Fig 1. The `quoted` manipulator

Streams, in contrast, are very easily extensible. Consumer manipulators can be defined to achieve the same functionality of `scanf()` and more. Going back to the last example, three manipulators `eat_int`, `eat` and `scanset` could be used to achieve the same effect as follows:

```
    istream is;
    char buffer[BUF_LEN];
    ...
```

```
is >> eat_int >> eat("BEGIN")
   >> scanset("A-Z", buffer);
```

Discussion

The C++ iostream package contains a rather small handful of pre-defined producer/consumer manipulators, the only instance of a consumer being the white-space eater `ws`. Other pre-defined manipulators set stream state variables which influence processing of input and output e.g. `hex`. The implementation of these manipulators relies on state in the implementation of the `ios` class as well as the implementations of the insertion and extraction operators corresponding to the affected types.

In this paper it has been demonstrated that producer and consumer manipulators can be used to enhance stream I/O formatting in a clean, easily extensible, type-safe fashion which is free from side-effects.

It is felt that producer and consumer manipulators offer a significant amount of as yet untapped potential. Considering the convenience, elegance, and safety they offer, it is perhaps rather surprising that the standard set of predefined manipulators (see Stroustrup sec 10.4.2.1 or AT&T C++ Programmer's Guide sec 5.8) is so limited in scope.

Acknowledgements

The idea of manipulators is due to Andrew Koenig.

I am grateful to Ian Utting for revising the paper and for making many useful suggestions.

References

B. Stroustrup, *The C++ Programming Language, Second Edition*, Addison Wesley, Reading, Massachusetts (1991).

SunPro, *C++ 3.0.1 Programmers Guide*, Sun Microsystems Inc, Mountain View, California (1992).

Note: Sources for many of the manipulators described in this paper and others are available by anonymous ftp from `eagle.ukc.ac.uk` in `pub/mr3/manip` (for GNU g++ 2.4.5 with libg++ 2.4).