Implementing Manipulators with the Standard IOStreams (Part I)

The last two installments of this column [1] were devoted to the implementation of stream inserters and extractors for user-defined types. Providing overloaded versions of the shift operators as input and output operations is not an entirely new technique and has been used by countless C++ programmers since the advent of C++ and its classic streams library. Yet we explored quite a number of new features in the standard IOStreams that aid implementation of more sophisticated and powerful inserters and extractors for user-defined types.

This time, we want to take a look at another category of abstractions related to IOStreams, which are nearly as important as inserters and extractors: the so-called *manipulators*. Much like inserters and extractors, user-defined manipulators could already be implemented using the different pre-standard IOStreams versions, but things are little different in the context of the standard IOStreams. In this and the subsequent column we will explain the general idea of manipulators for those who not familiar with them, and we will discuss what is new in standard IOStreams with regard to the implementation of manipulators.

A Short Recap

Manipulators are objects that can be inserted into or extracted from a stream in order to manipulate that stream. The standard IOStreams comes already with a number of predefined manipulators. We will henceforth call them the *standard manipulators*, in contrast to *user-defined manipulators.*

Here is a typical example of two standard manipulators:

```
cout << setw(10) << 10.55 << endl;
```

The inserted objects `setw(10)` and `endl` are the manipulators. The manipulator `setw(10)` sets the stream's field width to `10`; the manipulator `endl` inserts the end of line character and flushes the output. As you can see, manipulators can take arguments or be parameter-less.

Manipulators are inserted and extracted via shift operators. To be inserted and extracted this way, there must be shift operators defined for each type of manipulator. We will denote the type of a manipulator object as `manipT` in the following text. The inserter for a manipulator of type `manipT` looks like this:

```
template <class charT, class Traits>
basic istream<charT,Traits>&
operator<< (basic_istream<charT,Traits>& istr, const manipT& manip)
{ /*
   ... do the manipulation ...
  */
  return istr;
}
```

With this inserter defined, you can insert a manipulator object `Manip` of type `manipT` by saying:

```
cin << Manip;
```

The manipulator extractor is analogous.

The manipulation performed by a manipulator inserter or extractor can be any sequence of operations that lead to the desired manipulation of the stream. This sequence of operations can be encapsulated into a function, which we will call the associated function $f_{manipT}()$ in the following. There are several ways to associate a manipulator type `manipT` and its function $f_{manipT}()$. It turns out that there is a fundamentally different technique for manipulators with parameter than is used for manipulators without parameters. Some implementation idioms for manipulators can be studied by examining the standard manipulators. Additional techniques might be needed for advanced user-defined manipulators. Let's start with the simpler case of manipulators without parameters.

Manipulators Without Parameters

Parameter-less manipulators are the simplest form of manipulator: they are function pointers, and the manipulation is invocation of the function pointed to.

In IOStreams, the following function pointer types serve as manipulator types:

```
(1) ios_base&                   (*pf)(ios_base&)
```

```
(2) basic_ios<charT,Traits>&     (*pf)(basic_ios<charT,Traits>)
(3) basic_istream<charT,Traits>& (*pf)(basic_istream<charT,Traits>)
(4) basic_ostream<charT,Traits>& (*pf)(basic_ostream<charT,Traits>)
```

For these four manipulator types IOStreams already contains the required inserters and extractors. The extractor for parameter-less manipulators to input streams, for instance, takes the following form:

```
template<class charT, class traits>
basic_istream<charT, traits>& basic_istream<charT, traits>::operator>>
( basic_istream<charT,traits>& (*pf)(basic_istream<charT,traits>&) )
{  return (*pf)(*this); }
```

It uses a function pointer of type (3) from the list above. Similarly, an inserter for parameter-less manipulators to output streams uses a function pointer of type (4) and is already defined in IOStreams as:

```
template<class charT, class traits>
basic_ostream<charT, traits>& basic_ostream<charT, traits>::operator<<
( basic_ostream<charT, traits>& (*pf)(basic_ostream<charT, traits>&) )
{  return (*pf)(*this);  }
```

The inserters and extractors for function pointers of type (1) and (2) are also predefined in IOStreams. They allow insertion and extraction of parameter-less manipulator that can be applied to both input and output streams.

The list of manipulator types is not limited to the examples above. If you have created your own user-defined stream types, you might want to use additional signatures as parameter-less manipulators.

How to Implement Manipulators Without Parameters

Let's look at the standard manipulator `endl` as an example of a manipulator without parameters. The manipulator `endl`, which can be applied solely to output streams, is a pointer to the following function of type (4):

```
template<class charT, class traits>
inline basic_ostream<charT, traits>& endl(basic_ostream<charT, traits>& os)
{
  os.put( os.widen('\n') );
  os.flush();

  return os;
}
```

Hence an expression like:

```
cout << endl;
```

results in a call to the inserter:

```
ostream& ostream::operator<< (ostream& (*pf)(ostream&))
```

with `endl` as the actual argument for `pf`.

The standard contains another manipulator, `boolalpha`, that can be applied to input *and* output streams. Hence `boolalpha` is a pointer to a function of type `(1)`:

```
ios_base& boolalpha(ios_base& strm)
{
  strm.setf(ios_base::boolalpha);

  return strm;
}
```

In principle, every function that takes a reference to an `ios_base`, a `basic_ios`, a `basic_ostream`, or a `basic_istream`, and returns a reference to the same stream, can be used as a parameter-less manipulator.

Manipulators With Parameters

Manipulators with parameters are objects that can be inserted into and extracted from a stream via shift operators and take arguments that parameterize the manipulation of the stream. Insertion of a manipulator with one parameter would look like this:

```
cout << Manip(x);
```

The expression `Manip(x)` evaluates to a manipulator object of a manipulator type `manipT`, for which an according shift operator is defined. Manipulators with n parameters take the form `Manip(`x_1`,`x_2`,`x_3`,...,`x_n`)`. We restrict all consideration in this section to the case of manipulators with one argument. All presented solutions can canonically be applied to the case of any arbitrary number of arguments.

In the remaining part of this article we discuss a simple, straightforward way of implementing manipulators with parameters. In the next column we will refine this technique by factoring out the functionality that is common to all manipulators with parameters so that implementation of new manipulators becomes much easier because the common functionality can be reused. This time, however, we take a simple approach for explaining how manipulators with parameters can be implemented.

How to Implement Manipulators With Parameters

As a case study we implement a manipulator called `width`. It's effect is exactly the one produced by the standard manipulator `setw`: it sets the stream's field width when inserted to an output stream. The manipulator `width` is an example of a manipulator with one parameter that can be inserted into output streams only. Here is a possible implementation:

```
class width {
public:
   explicit width(unsigned int i) : i_(i) {}
private:
   unsigned int i_;

   template <class charT, class Traits>
   friend basic_ostream<charT,Traits>& operator<<
   (basic_ostream<charT,Traits>& ib, const width& w)
   {
     // the manipulation: set the stream's width
     ib.width(w.i_);
     return ib;
   }
};
```

The class type `width` is the manipulator type, i.e. it is a concrete example of the type we previously denoted as `manipT`. A width manipulator can be used like this:

```
cout << width(5) << 0.1 << endl;
```

Here, the manipulator expression `width(5)` is the construction of an unnamed object[1] of the manipulator type `width`. The argument that is passed to the manipulator expression is used to initialize the private data member `i_` of class `width`. This data member is later used by the shift operator for setting the field width.

Let us consider a second example: a multi-end-of-line manipulator, pretty much like the standard manipulator `endl`, but with the additional capability of inserting an arbitrary number of end-of-line characters. We will call it `mendl` and would want to use it like this:

```
cout  << mendl(5);
```

The implementation of `mendl` would look like this:

```
class mendl {
public:
   explicit mendl(unsigned int i) : i_(i) {}
private:
   unsigned int i_;

   template <class charT, class Traits>
   friend basic_ostream<charT,Traits>& operator<<
   (basic_ostream<charT,Traits>& os, const mendl& w)
   {
     // the manipulation: insert end-of-line characters and flush
     for (int i=0; i<i_;  i++)
        os.put(os.widen('\n'));
     os.flush();
   }
};
```

[1] While this is correct C++, some compilers are not capable of creating unnamed objects in conjunction with overloaded operators. We will show a possible work around in our next column.

Conclusion

In this column we discussed how manipulators work in general and how manipulators without parameters can be implemented using the predefined manipulator types and shift operators from the standard library. Additionally we had a look at a straightforward technique which allows the implementation of manipulators with parameters. In the next column we are going to refine this technique. We will also discuss the solution chosen for the standard manipulators in IOStreams.

References

[1] Klaus Kreft & Angelika Langer
User-Defined Inserters and Extractors – Part 1 & 2
C++ Report, September 1999 & January 2000