A More Refined Method for Implementing Manipulators with Parameters

In our last column [1] we started to look at manipulators in conjunction with the standard IOStreams. We had an in depth discussion of manipulators without parameters. Additionally we briefly glanced at manipulators with parameters by presenting a simple, straightforward technique to implement them. This time we refine this technique by factoring out the functionality that is common to al manipulators with parameters. We discuss possible variations in the implementation of manipulators with parameters, and eventually we explain the solution chosen for the standard manipulators in IOStreams. As in the last column, we will restrict our considerations to the case of manipulators with one argument; all presented solutions can canonically be applied to the case of any arbitrary number of arguments.

A Short Recap

In our last column we implemented a one-argument manipulator: `mendl`.

```
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();
   }
};
```

It is 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. To insert five times the end-of-line character into `cout` and than flush it, we can use `mendl` in the following way:

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

The manipulator expression `mendl(5)` is the construction of an unnamed object[1] of the manipulator type `mendl`. The argument that is passed to the manipulator expression is used to initialize the private data member `i_` of class `mendl`. `mendl`'s inserter uses this data member as the upper limit of the `for`-loop in which it inserts the end-of-line characters into the stream. To access this data member the `mendl`'s inserter is declared friend of `mendl`.

In a similar way we implemented a manipulator `width`. Its 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 main difference is the type of the stream that should be manipulated. It is the first argument of the shift operator for manipulator types. In the implementation of the manipulator `mendl` above, the first parameter of the inserter was of type `basic_ostream<charT,Traits>&`. For the `width` manipulator it is of type `ios_base&` instead. This difference stems from the fact that the stream functionality to manipulate the field width is defined higher up in the hierarchy (in class `ios_base`), whereas the functionality to insert a character or flush a stream is lower down (in class template `basic_ostream`). We will see later that this can make a significant difference for the implementation and use of amanipulator.

Introducing a Manipulator Base Class Template

Manipulators with one parameter may vary with respect to the type of their parameter and their respective functionality. We now build a framework that abstracts from those two properties and generally eases the implementation of manipulators with arbitrary functionality and parameters of arbitrary type.

The key idea is that the manipulation must not be hard-coded into the shift operators, but is factored out into an associated function. A pointer to this function is provided to the constructor of the manipulator object. It is then stored as a data member of the newly constructed object for subsequent invocation in the shift operator.

[1] Note that there are compilers that do not understand unnamed objects which are arguments to overloaded operators. If our compiler has this deficiency we must find a work-around, which will be discussed later in this article.

The manipulator type, which was a class type in the straightforward solution, now becomes a class template that takes the types of the manipulator arguments as template parameters. Concrete manipulator types are derived from this base class template.

Here is the suggested base class template called `one_arg_manip`:

```
template <class Argument>
class one_arg_manip
{
public:
 typedef void (* manipFct)(ios_base&, Argument);
 one_arg_manip(manipFct pf, const Argument& arg)
 : pf_(pf), arg_(arg) {}

private:
 manipFct pf_;
 const Argument arg_;

template <class charT, class Traits>
friend basic_istream<charT,Traits>& operator>>
(basic_istream<charT,Traits>& is, const one_arg_manip& oam)
{
 if (!is.good()) return is;
 (*(oam.pf_))(is,oam.arg_);
 return is;
}

template <class charT, class Traits>
friend basic_ostream<charT,Traits>& operator<<
(basic_ostream<charT,Traits>& os, const one_arg_manip& oam)
{
 if (!os.good()) return os;
 (*(oam.pf_))(os,oam.arg_);
 return os;
}
};
```

The core of the shift operators is invocation of the associated manipulator function with the manipulator arguments. Using the base manipulator template `one_arg_manip` the `width` manipulator from our last column could be re-implemented as follows:

```
class width : public one_arg_manip<unsigned int>
{public:
   explicit width(unsigned int i) : one_arg_manip<unsigned int>(width::fct, i)
  { }
 private:
  static void fct (ios_base& ib, unsigned int i)
  { ib.width(i); }
};
```

The manipulator type `width` is a subclass of the manipulator base template `one_arg_manip` instantiated on the type of the manipulator argument, which is `unsigned int` in this case. The manipulation, that previously was part of the shift operator, is now factored out into a static member function, called `fct()`, of class `width`.

When The Stream Type Matters

Note that the associated manipulator function has a particular function signature: it takes a reference to the stream base class `ios_base`. A manipulator, however, might need information or functionality that is specific to a particular stream type and not accessible via the stream base class `ios_base`. Consider for instance the multi-end-of-line manipulator `mendl`. We have already mentioned above that it calls member functions that are specific to `basic_ostream`, like `put()` and `flush()`. If you build it using the manipulator base template technique, which we have just seen, then the manipulator type itself will be a class template. Let us see why.

First, the manipulator function `fct` becomes a function template, because the stream it operates on (which is `basic_ostream`) is a class template. Inevitably, the manipulator type `mendl` and the manipulator base class `one_arg_manip` become templates, too. As a consequence we get the following implementation for `mendl`, assuming that the function signature in `one_arg_manip` has also changed as necessary:

```
  template <class charT, class Traits = char_traits<charT> >
  class mendl : public one_arg_manip<charT,Traits,unsigned int>
  {public:
    explicit mendl(unsigned int i)
    : one_arg_manip< charT,Traits, unsigned int>(mendl::fct,i) { }
   private:
    static void fct(basic_ostream<charT,Traits>& os, unsigned int n)
    { for (int i=0; i<n;  i++)
          os.put(os.widen('\n'));
      os.flush();
    }
  };
```

Now that `mendl` is a template, the manipulator expression is less convenient than it used to be. Each time we manipulate a stream by inserting a `mendl` object, we need to know the character (and traits type) of that stream. Instead of simply saying:

```
  cout  << mendl(5); // wrong ! – does not compile
  wcout << mendl(5); // wrong ! – does not compile
```

we now have to specify the template arguments and say:

```
  cout  << mendl<char>(5);
  wcout << mendl<wchar_t>(5);
```

Note that this would not have happened with a pre-standard IOStreams, where all streams are narrow character streams. All stream functionality such as transporting characters and formatting and parsing character sequences is only available for characters of type `char` in a pre-standard IOStreams. Hence *all* stream classes - not only the root of the class hierarchy - were simple classes and not templates. In the pre-standard context it was therefore sufficient that the manipulators with parameters were simple classes, too.

As we are using the standard IOStreams here, we need to find possibilities to limit the inconvenience. We can do this by defining typedefs for each of the instantiations of `mendl`. Then we have different manipulators for each type of stream. It is an improvement because we need not know the character and traits type of the stream. Instead we have to figure out which manipulator type is the right one to be used with a particular stream:

```
  typedef mendl<char>    nmendl;
  typedef mendl<wchar_t> wmendl;

  cout  << nmendl(5);
  wcout << wmendl(5);
```

Note that `nmendl` and its usability are equivalent to those of a manipulator implemented for a pre-standard IOStreams.

Another idea for making the manipulator calls more convenient relies on automatic function template argument deduction: The compiler is capable of deducing function template arguments from the actual arguments provided to a function call. We can use this language feature to have the compiler figure out the character and traits type of a stream and the respective manipulator: We wrap the construction of a manipulator object into a function that we call `mendl` and rename the manipulator type to `basic_mendl`. In other words, we add the following function template:

```
  template <class charT, class Traits>
  basic_mendl<charT,Traits> mendl(unsigned int n, basic_ostream<charT,Traits>&)
  { return basic_mendl<charT,Traits>(n); }
```

The manipulator expression would now be a call to the wrapper function instead of the construction of an unnamed object of the manipulator type. We would use the `mendl` function like this:

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

The downside is that we have to redundantly repeat the stream object in the manipulator expression.

This example uses an implementation technique that might be also a sensible alternative if your compiler is not capable of creating unnamed objects in conjunction with overloaded operators. When we introduced the straightforward implementation of `mendl` at the beginning of the column we explained that an unnamed object of the manipulator type is constructed to which an appropriate shift operator is applied. Some compilers have problems with unnamed objects that appear in the context of overloaded operators. For those compiler it is an alternative to implement the manipulator as a function which returns the manipulator object. And that is exactly what we did in the example above where we implemented the manipulator as a manipulator function `mendl` which returns a manipulator object of type `basic_mendl`.

Now that we have seen some alternative implementations of manipulators with parameters, let us see how it is done in the standard IOStreams library for the standard manipulators.

Manipulators With Parameters in The Standard IOStreams

The IOStreams library contains a number of predefined standard manipulators with parameters, such as `setw()`, `setfill()`, etc. . They are implemented as functions that return an object of a type called *`smanip`*. The name *`smanip`* does not denote an actual type in the library, but is a placeholder for one or more implementation-specific types. Each standard manipulator is allowed to return an object of a different type. The standard does not specify any details regarding this/these type(s). It is likely that in most implementations *`smanip`* is a class template very similar to `one_arg_manip`.

When you aim to implement a manipulator with a parameter, your first impulse might be to take a look at the implementation of a standard manipulator in the library and reuse the base manipulator template *`smanip`* from the library. You might find that a standard manipulator like `setw` is a function that returns an object of a particular *`smanip`* type, e.g. called `smanip<unsigned int>` or `_Smanip<int>`. It is advisable not to reuse the *`smanip`* type(s), because they are implementation-specific and using them makes your manipulator implementation non-portable. Instead, implement your own manipulator base template similar to `one_arg_manip`, use that for building user-defined manipulators, and stay independent of library specifics.

Wrap-Up

In this and the previous column we have explored techniques for implementation of manipulators, which are object that can be inserted to or extracted from a stream and manipulate the stream as a side effect of the insertion or extraction. The manipulation can be as simple as inserting an end-of-line character and flushing the stream (see the standard manipulator `endl`), but user-defined manipulators can potentially perform arbitrarily complex manipulations. Especially the parameterized manipulators can be really powerful abstractions. The intent of the columns was not to demonstrate the power of manipulators (we leave that as an exercise to our creative readers), but to suggest implementation techniques using the standard IOStreams.

References

[1] Klaus Kreft & Angelika Langer
Implementing Manipulators Using the Standard IOStreams
C++ Report, April 2000