User-Defined Inserters and Extractors Revisited

In our last column [1], we started discussing the implementation of user-defined inserters and extractors, that is, operators that allow insertion of objects of a user-defined type into an output stream or extraction of such objects from an input stream. This time we aim to improve the inserters and extractors and we will show you a significantly more sophisticated implementation.

For implementation of user-defined inserters and extractors we studied a simple, yet typical, approach: decomposition of the type (for which an inserter and extractor are implemented) into its parts and use of existing input and output operations for these parts. As an example for this we used a class `date`[1] defined as:

```
class date
{
   public:
      date(int d, int m, int y);
      date(const tm& t);
      date();
      // more constructors and useful member functions
    private:
      tm tm_date;
};
```

and implemented its inserter and extractor as:

```
template<class charT, class Traits>
basic_ostream<charT, Traits>&
operator<< (basic_ostream<charT, Traits >& os, const date& dat)
{
 os << dat.tm_date.tm_mon << ' ';
 os << dat.tm_date.tm_mday << ' ';
 os << dat.tm_date.tm_year ;
 return os;
}

template<class charT, class Traits>
basic_istream<charT, Traits>&
operator>> (basic_istream<charT,Traits>& is, date& dat)
{
 is >> dat.tm_date.tm_mday;
 is >> dat.tm_date.tm_mon;
 is >> dat.tm_date.tm_year;
 return is;
}
```

These definitions allow use of a date object together with any input and output stream. While this approach is sufficient in many situations, we discussed areas where it can be improved. These areas are:

Internationalization
The textual representation of a date value varies among cultural areas. The inserter and extractor from the example above, however, ignore this fact and are incapable of adjusting the formatting and parsing of dates to cultural conventions. The standard library offers solutions to some of these culture dependent parsing and formatting issues (see [2]). Culture-sensitive parsing and formatting is factored out into exchangeable components: locales and facets. For instance, the standard facets `time_put` and

[1] The type `tm` used for the private data member of `date` is a structure defined in the C library (in header file `<ctime>`). It is a type suitable for representing date values and consists of a number of integral values, among them the day of a month, the month of a year, and the year. Note that the way `tm` is used by the simple inserter and extractor does not conform to `tm` conventions imposed by the C library. The conventions, for instance, require that the number representing the month must be between 0 and 11; otherwise the function working on `tm` structures do work properly. We can simple ignore these rules in our example because we use the `tm` structure just as a data store, but not in any `tm`-typical way.

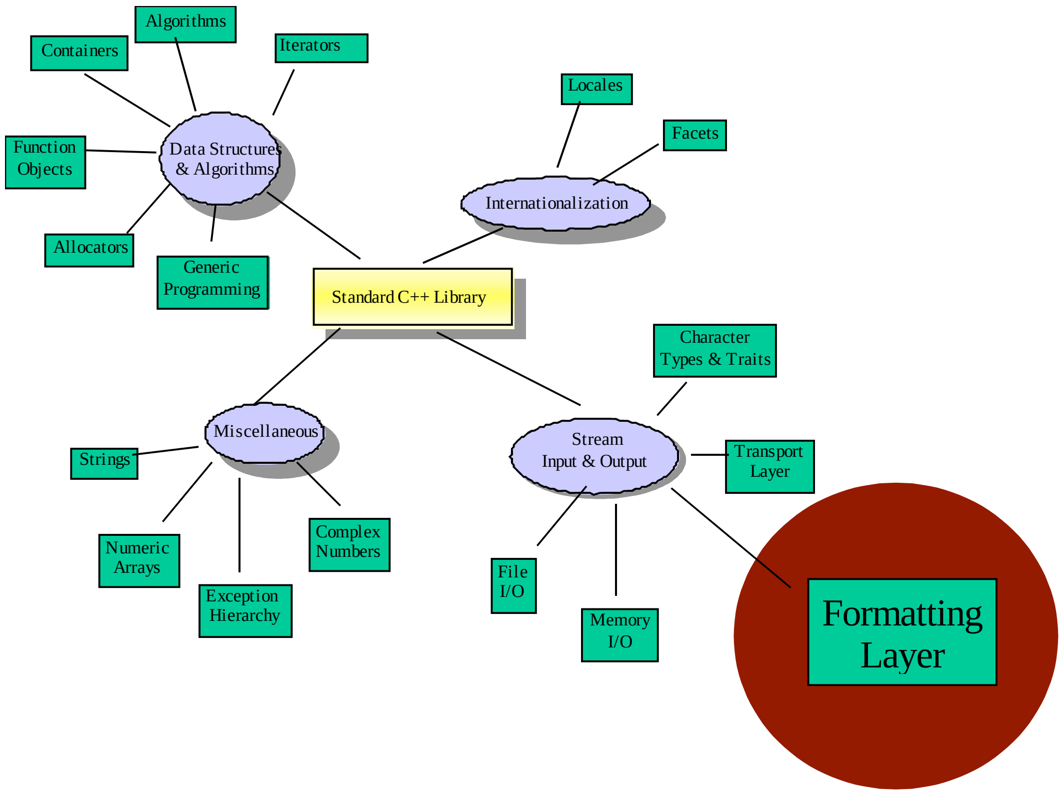
`time_get` represent the knowledge about date and time formats. In our example above, we might want to make use of these culture-sensitive parsing and formatting services.

Format Control
Our simple inserter from the example above has a problem with the field adjustment. If the field width is set to a particular value, only the first item printed would be adjusted properly, because the first inserter will reset the field width to zero. Probably the expected result after setting the field width prior to insertion of a date object is that the entire date is adjusted, and not just the first part of it. You might want to fix this problem and control the field width yourself. This leads us to the more general problem of format control in inserters and extractors. For sake of consistency, format control facilities defined in IOStreams should generally be interpreted and manipulated by user-defined i/o operations in the same way as the predefined inserters and extractors do it.

Prefix and Suffix Operations
Typical prefix activities for an extractor are flushing of a tied stream and skipping of white spaces before the real value is extracted. In our example, these activities are performed repeatedly, although they were necessary only once, before the actual output of a date happens.
In IOStreams, the prefix and suffix activities are encapsulated into classes called `sentry`, nested into the stream class templates `basic_istream` (for extractors) and `basic_ostream`. (for inserters). The constructors of these classes perform the prefix activities, the destructors carry out the suffix activities. Our example could be improved so that is performs the prefix and suffix activities once per output of a date object.

Error Indication
Errors can occur during parsing or formatting of an item. Consider the extractor from the example above: we might want to check the extracted date's validity and indicate failure if the date is incomplete or is February 31 for instance. Further errors can occur within functions used by the inserter or extractor and can lead to failure of the operation. Users of inserters and extractors in IOStreams expect that error situations are indicated by means of the stream state and by throwing exceptions according to the stream’s exception mask. The predefined i/o operations for built-in and library types demonstrate the principle.

In this installment of the column, we implement improved versions of the date's inserter and extractor by gradually adding refinements in those areas discusses above. So, again the focus is on the formatting facilities of standard IOStreams (see diagram below).



Above, we already sketched out several areas of improvement for the date's inserter and extractor. Let us start with internationalization.

Internationalization

In our last column, we suggested a number of steps to take for implementation of inserters and extractions of culture-sensitive data types. These were:
(1) Identify the culture dependencies that are related to the respective data type.
(2) Encapsulate relevant culture-dependent rules and services into facets.
(3) Provide locales that contain such facets, or provide means for creating such extended locales.
(4) Imbue streams with such locales.
(5) Use the stream's locale and its facets for implementing your internationalized inserters and extractors.

Let us see, how these steps apply to our example of date formatting. The format of dates is clearly culture-dependent. Consequently, we need facets that represent the rules for formatting and parsing of dates. Fortunately, the standard library already contains the facets `time_put` and `time_get` for this purpose. As a result, we need not create new facet types, but can use standard facets instead.[2] Also, we need not care about equipping locales with the necessary facets or imbuing streams with such extended locales because every locale object contains at least all standard facets. We can concentrate on the step (5): use of facets for implementing an internationalized version of our date inserter and extractor. Here is the internationalized version of the inserter and extractor:

[2] Note that by the time we will be using the standard time facets, we must adjust our `date` class so that it uses the `tm` structure according to the C library conventions. We would for instance re-implement the constructor as follows:

```
date(int d, int m, int y)
   { tm_date.tm_mday = d; tm_date.tm_mon = m-1; tm_date.tm_year = y-1900;
     tm_date.tm_sec = tm_date.tm_min = tm_date.tm_hour = 0;
     tm_date.tm_wday = tm_date.tm_yday = 0;
     tm_date.tm_isdst = 0;
   }
```

The extractor:

```
template<class charT, class Traits>
basic_istream<charT, Traits>&
operator>> (basic_istream<charT, Traits >& is, date& dat)
{
 ios_base::iostate err = 0;

 use_facet<time_get<charT,istreambuf_iterator<charT,Traits> > >(is.getloc())
   .get_date(is, istreambuf_iterator<charT,Traits>(),is, err, &dat.tm_date);

 return is;
}
```

The inserter:

```
template<class charT, class Traits>
basic_ostream<charT, Traits>&
operator<< (basic_ostream<charT, Traits >& os, const date& dat)
{
 use_facet<time_put<charT,ostreambuf_iterator<charT,Traits> > >(os.getloc())
   .put(os,os,os.fill(),&dat.tm_date,'x');
 return os;
}
```

Use of the `time_get` and `time_put` facets is similar to use of the other numeric and monetary parsing and formatting facets. We will not go into the details here, but at least give you a rough idea of the time facets' interfaces. A time formatting facet of type `time_put<char_type,iter_type>` has the following function for formatting time and date values:

```
iter_type put(iter_type out
            , ios_base& fmt, char_type fill
            , const tm* time
            , char fmtspec, char fmtmodifier = 0) const;
```

A time parsing facet of type `time_get<char_type,iter_type>` has the following parsing function for dates:

```
iter_type get_date(iter_type in, iter_type end
            , ios_base& fmt
            , ios_base::iostate& err
            , tm* time) const;
```

Here is a brief description of the function arguments:

- *Iterators*. Formatting functions like `put()` take an output iterator (parameter `out`) that designates the destination for output. Parsing functions like `get_date()` take an input iterator range (parameters `in` and `end`) that designates begin and end of the character sequence to be parsed. The operations return an iterator that points to the position after the sequence written to or read from.
  The iterators used in our example are input and output stream buffer iterators. The begin iterators are created by converting the stream itself into stream buffer iterators. This is possible because the `istreambuf_iterator` and the `ostreambuf_iterator` have converting constructors that take a reference to a stream and convert it into a stream buffer iterator to the current stream position. The end iterator is created my means of the default constructor of class `istreambuf_iterator`.
- *Formatting information*. Different from other formatting facets in the locale, the time facets neither use any format information (such as the field width) from the `ios_base&` object that is provided as an argument (parameter `fmt`) nor do they use the provided fill character (parameter `fill`). We provide these arguments anyway, because non-standard time facets might be using them.
- *Value*. Naturally, parsing and formatting functions take a pointer or reference to the value to be written or read. In this case it's a pointer to a time structure (parameter `time`).

The interface of our `date` class would, however, not change in any way and for this reason we omit these details.

- *Format specification*. The time formatting function takes a format specifier, plus an optional format modifier (parameters `fmtspec` and `fmtmodifier`). These are characters as defined for the C library function `strftime`() (defined in header `<ctime>`).
- *Error indication*. Parsing functions like `get_date()` store error information in an `ios_base::iostate` object (parameter `err`). Formatting functions like `put()` do not have an error parameter. If the output iterator returned by the `put()` function has a `failed()` member function then you can use this for checking the success or failure.

Prefix and Suffix Operations

We will now add the prefix and suffix operations by means of sentries. As a reminder, here are the recommendations for use of `sentry` classes in inserters and extractors:
(1) Create a sentry object prior to any other activity.
(2) Check for success of the prefix operations after construction of the sentry object by means of its `bool` operator. Return from the function if the check after construction of the sentry object does not indicate success.

Below you find the previous example, extended by use of sentries.

The extractor:
```
template<class charT, class Traits>
basic_istream<charT, Traits>&
operator>> (basic_istream<charT, Traits >& is, date& dat)
{
 ios_base::iostate err = 0;
 typename basic_istream<charT,Traits>::sentry ipfx(is);
 if(ipfx)
 {
   use_facet<time_get<charT,istreambuf_iterator<charT,Traits> > >(is.getloc())
     .get_date(is, istreambuf_iterator<charT,Traits>(),is, err, &dat.tm_date);
 }
 return is;
}
```
The inserter:
```
template<class charT, class Traits>
basic_ostream<charT, Traits>&
operator<< (basic_ostream<charT, Traits >& os, const date& dat)
{
 typename basic_ostream<charT,Traits>::sentry opfx(os);
 if(opfx)
 {
   use_facet<time_put<charT,ostreambuf_iterator<charT,Traits> > >(os.getloc())
     .put(os,os,os.fill(),&dat.tm_date,'x');
 }
 return os;
}
```
Compared to the original example, where all parts of the date were printed separately and with each part the prefix and suffix operations were invoked, the improved version shown above triggers the prefix and suffix once per output of a date value.

Format Control

Let us now add format control to the inserter and extractor. The only format control parameter that we want to add is proper use of the field width when date objects are written to output streams. This includes that we take into account the stream's field width setting, but also the adjustment flags (`right`, `left`, `internal`) and the fill character.
Naturally, we intend to follow the recommendations that we previously explained:
(1) As the field width is not permanent, but is reset to 0 each time it was user, we need to reset the field width to 0 at the end of our inserter.
(2) The `adjustfield` need not be initialized, so we stick to the rule that all inserters behave as though the field were set to `right` unless the `adjustfield` is set to any other value.

Below you find the inserter from the previous example, with field width adjustment added.

The inserter:

```
template<class charT, class Traits>
basic_ostream<charT, Traits>&
operator<< (basic_ostream<charT, Traits >& os, const date& dat)
{
 if (!os.good()) return os;

 typename basic_ostream<charT,Traits>::sentry opfx(os);
 if(opfx)
 {
   basic_stringbuf<charT,Traits> sb;

   use_facet<time_put<charT,ostreambuf_iterator<charT,Traits> > >(os.getloc())
     .put(&sb,os,os.fill(),&dat.tm_date,'x');

   basic_string<charT, Traits> s = sb.str();
   streamsize charToPad = os.width() - s.length();
   ostreambuf_iterator<charT,Traits> sink(os);
   if (charToPad <= 0)
      {
        sink = copy(s.begin(), s.end(), sink);
      }
   else
      { if (os.flags() & ios_base::left)
           { sink = copy(s.begin(), s.end(), sink);
             sink = fill_n(sink,charToPad,os.fill());
           }
        else
           { sink = fill_n(sink,charToPad,os.fill());
             sink = copy(s.begin(), s.end(), sink);
           }
      }
 }
 os.width(0);
 return os;
}
```

Error Indication

Finally, we add error handling to the inserter and extractor. Here are the recommendations that were explained in the previous column:

In order to detect error situations:

- Catch exceptions.
- Check return codes, the stream state, or other error indications.
- Check the validity of extracted objects, and determine other potential errors.

In order to indicate error situations. Have your inserter and extractor report errors according to the IOStreams principles:

(a) Set the stream state flags as follows:
   - `ios_base::badbit` to indicate loss of integrity of the stream.
   - `ios_base::failbit` if the formatting or parsing itself fails due to the internal logic of your operation.
   - `ios_base::eofbit` when the end of the input is reached.

(a) Raise an exception if the exception mask asks for it.
If any of the invoked operations raises an exception, catch the exception and rethrow it, if the exception mask allows it.

First, we extend the date class and add a `bool` operator to the `date` class, that checks the validity of the date. This way we can also demonstrate error situations that are due to extraction of invalid objects. Here is the complete declaration of the extended date class:

```
class date {
public:
   date(int d, int m, int y);
   date(tm t);
   date();
   bool operator!();            // check for the date's validity

private:
   tm tm_date;

template<class charT, Traits>
friend basic_istream<charT, Traits>&
operator>> (basic_istream<charT, Traits >& is, date& dat);

template<class charT, Traits>
friend basic_ostream<charT, Traits>&
operator<< (basic_ostream<charT, Traits >& os, const date& dat);
};
```

The `bool` operator is used in the extractor below to check whether the extracted date is valid. Here are the inserter and extractor, this time with error handling and error indication added:

The extractor:

```
template<class charT, class Traits>
basic_istream<charT, Traits>&
operator>> (basic_istream<charT, Traits >& is, date& dat)
{
 ios_base::iostate err = 0;
 try
 {
   typename basic_istream<charT,Traits>::sentry ipfx(is);
   if(ipfx)
   {
    use_facet<time_get<charT,istreambuf_iterator<charT,Traits> > >
      (is.getloc()).get_date
         (is, istreambuf_iterator<charT,Traits>(),is, err, &dat.tm_date);

    // check for the date's validity
    if (!dat) err |= ios_base::failbit;

   }
 }
 catch(bad_alloc& )
 {
  err |= ios_base::badbit;
  ios_base::iostate exception_mask = is.exceptions();

  if (    (exception_mask & ios_base::failbit)
     &&  !(exception_mask & ios_base::badbit ))
  {
     is.setstate(err);
  }
  else if (exception_mask & ios_base::badbit)
  {
     try { is.setstate(err); }
     catch( ios_base::failure& ) { }
     throw;
  }
 }
 catch(...)
 {
  err |= ios_base::failbit;
  ios_base::iostate exception_mask = is.exceptions();

  if (  (exception_mask & ios_base::badbit)
     && (err & ios_base::badbit))
  {
     is.setstate(err);
  }
  else if(exception_mask & ios_base::failbit)
  {
     try { is.setstate(err); }
     catch( ios_base::failure& ) { }
     throw;
  }
 }
 if ( err ) is.setstate(err);
 return is;
}
```

The inserter:

```
  template<class charT, class Traits>
  basic_ostream<charT, Traits>&
  operator<< (basic_ostream<charT, Traits >& os, const date& dat)
  {
   ios_base::iostate err = 0;
   try
   {
     typename basic_ostream<charT,Traits>::sentry opfx(os);
     if(opfx)
     {
        basic_stringbuf<charT,Traits> sb;

        // formatting the date
        if (use_facet<time_put<charT,ostreambuf_iterator<charT,Traits> > >
            (os.getloc()).put(&sb,os,os.fill(),&dat.tm_date,'x').failed()
           )
           // set the stream state after checking the return iterator
          err = ios_base::badbit;

        // field width adjustment
        if (err == ios_base::goodbit)
        {
          basic_string<charT, Traits> s = sb.str();
          streamsize charToPad = os.width() - s.length();
          ostreambuf_iterator<charT,Traits> sink(os);

          if (charToPad <= 0)
          {
            sink = copy(s.begin(), s.end(), sink);
          }
          else
          {
            if (os.flags() & ios_base::left)
            {
              sink = copy(s.begin(), s.end(), sink);
              sink = fill_n(sink,charToPad,os.fill());
            }
            else
            {
              sink = fill_n(sink,charToPad,os.fill());
              sink = copy(s.begin(), s.end(), sink);
            }
          }
          if (sink.failed())
             err = ios_base::failbit;
        }
        os.width(0);
     }
   }
   catch(bad_alloc& )
   {
    err |= ios_base::badbit;
    ios_base::iostate exception_mask = os.exceptions();
    if (   (exception_mask & ios_base::failbit)
       && !(exception_mask & ios_base::badbit) )
    {
       os.setstate(err);
    }
    else if (exception_mask & ios_base::badbit)
    {
       try { os.setstate(err); }
       catch( ios_base::failure& ) { }
       throw;
    }
   }
   catch(...)
   {
    err |= ios_base::failbit;
    ios_base::iostate exception_mask = os.exceptions();

    if (  (exception_mask & ios_base::badbit)
       && (err & ios_base::badbit))
    {
        os.setstate(err);
    }
    else if(exception_mask & ios_base::failbit)
    {
        try { os.setstate(err); }
        catch( ios_base::failure& ) { }
        throw;
    }
   }
   if ( err ) os.setstate(err);
   return os;
  }
```

As recommended, we dutifully keep track of all error situations by catching all exceptions, checking the stream state and other error indications, and detecting invalid dates. We then set the stream state and raise exceptions. Let us explain all of the considerations involved in greater detail below. As the issues are rather complex, we group the discussion by topics: manipulation of the stream state and the strategy for catching and handling the exceptions.

Setting the stream state.

Both the inserter and the extractor maintain a local stream state object `err`. The failure of any invoked operation is accumulated in this temporary stream state. (Note, that accumulation of errors is not required by the IOStreams framework. You can alternatively stop when the first error is detected.) Eventually the temporary stream state replaces the current stream state. Let us see how the various activities contribute to the stream state.

Extractor. In the extractor, the parsing operation of the `time_get` facet uses the stream state object for reporting its success or failure. The parsing operation might set any of the state flags as appropriate. We then check the extracted date's validity by means of its `bool` operator and add `failbit` to this stream state object if the date is invalid. We set `failbit` instead of `badbit` because we consider extraction of an invalid date a recoverable parsing failure (hence `failbit`) rather than a loss of the stream's integrity (which would have required setting `badbit`).

Inserter. In the inserter, the result of the formatting operation of the `time_put` facet is checked by calling the returned iterator's `failed`() function. In case of failure, we set the temporary stream state object to `badbit` instead of `failbit` because a failed formatting operation indicates a broken stream rather than a recoverable formatting failure.

The local stream state object is eventually used to adjust the stream's state if no exceptions have been raised so far. The stream state is set via the `setstate`() function, which takes a new stream state value, replaces the current stream state with the new value, and automatically raises an `ios_base::failure` exception if the exception mask requires it.

Catching and throwing exceptions.

Compared to the original solution, all relevant statements are now wrapped into a try block. The reason is that in an inserter and extractor we must catch all exceptions that are thrown by any invoked operation because we cannot simply let exceptions propagate out of an i/o operation. We must first check whether the exception mask permits the caught exception to be propagated outside the inserter or extractor. Only if the exception mask allows it we rethrow the exception, otherwise we suppress it. Additionally, we must set the stream state appropriately in order to reflect the error situation that was detected by catching an exception.

Exceptions caught in an i/o operation are best handled as follows:
(1) The caught exception is qualified as an indication of failure (equivalent to `failbit`) or of loss of integrity (equivalent to `badbit`).
(2) The exception mask is checked in order to find out whether an exception must be thrown.
(3) The stream state must be set and, if required, an exception must be raised.

In our example, both the inserter and the extractor handle caught exceptions exactly the same way. They both have two catch clauses: We qualify memory shortage indicated by a `bad_alloc` exception as a loss of the stream's integrity (`badbit`), and all other exceptions as failure of the operation (`failbit`). The respective error flag is added to the local stream state object.

By examining the exception mask and the local stream state object, we determine whether an exception must be raised at all and if so, which exception it must be: either `ios_base::failure` or the originally caught exception.
Due to the accumulation of errors, it can happen that both `badbit` and `failbit` are set in the local stream state object. In case that both flags are also set in the exception mask, we would have to raise two exceptions, so to speak. As only one exception can be thrown, we decide to throw the exception that belongs to the `badbit`, because we consider a `badbit` situation the more severe error situation. The exception associated to the `badbit` is `bad_alloc`, if such an exception was caught, or `ios_base::failure` otherwise.

After examination of the exception mask and the local stream state object, we now know whether we must update the stream state and raise an exception. This is done as follows:

The `setstate()` function is invoked with the accumulated local stream state as an argument. This call sets the stream state and might raise an `ios_base::failure` exception if the exception mask asks for it. In our case we do not want to throw an `ios_base::failure` exception, but the originally caught exception, so that we must suppress the exception thrown by the `setstate()` function. For this reason, the `setstate()` function is called in a `try` block with an empty corresponding `catch` block. As a result, the stream state is set to the accumulated local stream state, as intended, and the automatically raised `ios_base::failure` exception is caught and discarded. Then, the originally caught exception is rethrown.

Simple versus Refined Approach

As you can see, the implementation of the refined inserter and extractor is significantly more complex than the initial simple approach discussed in the last article, where we just decomposed the date object into its parts and used existing inserters and extractors for the parts. Let us spend a thought on the trade-off between the simple, straightforward and the more complex, refined implementation. When would we want to take one or the other approach?

There is no carved-in-stone rule for the "right" level of refinement for inserters and extractors. The degree of sophistication that must be implemented depends on the situation where these operators shall be used. For instance, if the inserter is predominantly used for writing trace output to a log file, then the simple approach might be fully sufficient. Conversely, the inserters might be intended to be used for formatting of objects by means of string streams, that is, for conversion between the object's binary representation and a human-readable string representation. If these string representations are to be displayed in an application that is designed for world wide use, then obviously more sophisticated operators are needed.

When we compare the simple inserter and extractor with the refined one, it is evident that these refinements do not come for free. Besides the greater effort for implementation of the refined inserter and the extractor, the refined approach needs significantly more time during the design stage. For the simple approach you just have to identify the parts that the user-defined type consists of and reuse existing operators. This is easy. For the refined approach, on the other hand, each of the areas of refinement needs a sound design. Take for example the error handling. You have to decide which error can occur. Then you have to determine to which error states (bad or failure) each error should be mapped. An equivalent number of considerations is necessary, if format control or internationalization shall be added.

As you could see in our example, both approaches differ significantly in the amount of code that is needed for their implementation. Accordingly, both approaches differ in the amount of time needed for the implementation and the performance of the resulting implementation. Let's discuss both issues, run time performance and implementation effort.

Run time performance. Operators for built-in types, which are defined in the standard IOStreams library for `basic_istream` and `basic_ostream`, are implemented in pretty much the same way that we suggested in our refined date inserter and extractor above. Almost everything in the standard C++ library is designed for efficiency and optimal run time performance, and indeed, the more sophisticated inserters and extractors can have better performance than the simple ones. If the simple approach is taken, then the user-defined type is decomposed into its parts and for each of the parts an existing inserter or extractor is invoked. Inevitably, certain functionality, for instance the prefix and postfix operations, are executed redundantly.

Implementation effort. Regarding the amount of time needed for implementation of a sophisticated inserter or extractor, it is definitely true that the refined approach requires a greater effort. The implementation effort can, however, be reduced substantially if the IOStreams specific refinements are implemented once, as a generic inserter and extractor, and the i/o-functionality for each user-defined type is just hooked into the generic operators.

Summary

In this column, we continued the discussion of inserters and extractors for user-defined types. We compared a straightforward, simple approach, based on decomposition of the user-defined type, to a more sophisticated implementation, that allows to address special requirements such as optimal performance, better error indication, adaptability to cultural conventions, and smarter format control.

References

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

[2] Klaus Kreft & Angelika Langer
The Standard Facets
C++ Report, November/December 1997