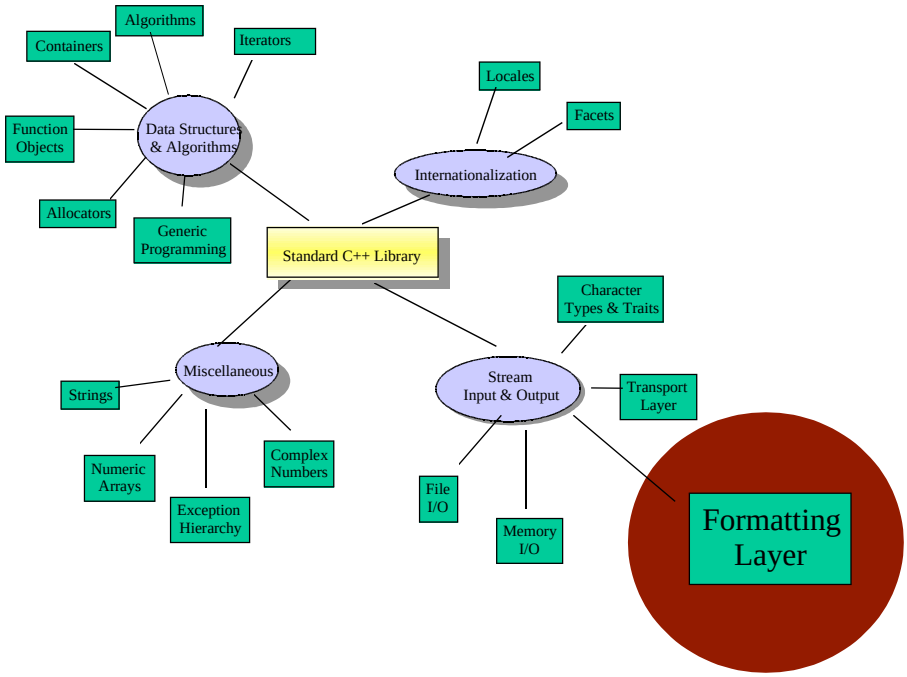
User-Defined Inserters and Extractors (Part I)

The past 3 installments of this column were devoted to various interesting iterators in the standard library, among them inserters and stream iterators. After this intensive discussion of iterators, it is now time to move on to another part of the standard library. In this and the next article we will take a look at IOStreams, or, to be more specific, we will discuss stream inserters and extractors (see diagram below).



What are inserters and extractors? They are also known as shift operators for streams, that is, they are overloaded versions of the bit shift operators << and >> and they allow insertion of an object into an output stream or extraction of an object from an input stream: Below is the classic "hello world" example that we've all tried out at least once in the early days of our life as C++ programmers:

```
cout << "hello world\n";
```

The inserter that is called here is an overloaded version of `operator<<()` for objects of type `const char*`. The standard library provides inserters and extractors for all built-in types and many of the types defined in the standard library such as complex numbers and strings. For instance, we can insert and extract integral values using the predefined shift operators as shown below:

```
int i = 5;
cout << i;   // insertion of an integer into an output stream
cin  >> i;   // extraction of an integer from an input stream
```

Most inserters for built-in types[1] such as `int` are member functions of the stream base class template `template <class charT, class traits> class basic_ostream`.. All output stream classes in the standard library are derived from `basic_ostream` and inherit these operations so that values of built-in types can be inserted into any kind of output stream. Similarly, the stream base class template `template <class charT, class traits> class basic_istream` contains the extractors for most of the built-in types.

As mentioned above, most of the abstractions contained in the standard library provide an inserter and an extractor for their own type. For instance, the standard library comes with overloaded versions of the shift operators for the string abstraction `template <class charT, class traits, class Allocator> class basic_string`. For other classes in the standard library, no inserter and/or extractor is predefined. Examples are all the STL containers. For these abstractions, the user must find his own way of inserting

[1] The exception from that rule are the inserters for characters and character sequences; they are implemented as friend functions.

and extracting objects of these types. We showed how this can be done by means of stream iterators and the `copy()` algorithm in one of our previous columns (see /1/ for the details).

The real challenge for a standard library user arises when a newly created user-defined type shall be used in conjunction with the iostreams and an inserter and an extractor for the new type must be implemented. In what follows we will discuss how such operations can be implemented in terms of an example - a user-defined `date` type. Say, the `date` type is defined as follows:

```
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;
};
```

The type `tm` used in the implementation above 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 since 1900.

The goal is to allow insertion and extraction of `date` objects in exactly the same way as input and output of built-in types, like in the following code fragments:

```
date theDate(9,9,1999);
cout << "funny date: " << theDate << '\n';
```

or

```
date aDate;
cout << '\n' << "Please, enter a date (day month year)" << '\n';
cin  >> aDate;
cout << aDate << '\n';
```

In order to explain the implementation of inserters and extractors for user-defined types, we first take a straightforward, relatively simple approach. We will evaluate it and see where it has weaknesses and deficiencies. Naturally, we will consider refinements to address the problems and in the next installment of this column, we will show you how the refinements can be implemented.

From the introduction above we already know that we have to implement an `operator<<()` as inserter and an `operator>>()` as extractor for the `date` class. Before we go into the implementation details of the actual functionality of these operators, let us first discuss their signature.

The Inserter's and Extractor's Function Signature

Inserters and extractors are function templates. This is because all stream types are class templates taking the character type and the character traits type as template arguments. When we implement inserters and extractors, then we want to enable input and output to all types of streams regardless of their character and traits type. For this reason, inserters and extractors are typically function templates that take as template arguments the character type and the character traits type of the stream they operate on.

Next, let us consider what the function arguments and the return value should be. Inserters and extractors must take a reference to the stream to/from which the data shall be inserted/extracted and they must return a reference to the same stream object. This is because shift operators can be chained and the return value of a shift operator must be usable as input to the next shift operator in an operator chain. Otherwise we would not be capable to allow the intended convenient notation such as:

```
date theDate(9,9,1999);
cout << "funny date: " << theDate << '\n';
```

The second function argument is a reference to an object of the user-defined type that is written or read. Inserters take a constant reference, because the inserter is not supposed to modify the object it prints; extractors take a non-constant reference, because they alter the object by filling it with information extracted from the stream.

One question is still left open: Of which type shall the stream argument and the return value be? The natural choice for the argument and return type of inserters and extractors are `basic_istream <class charT, class traits>` for extractors and `basic_ostream <class charT, class traits>` for inserters because these classes define the inserters and extractors for built-in types. It is certainly recommendable that user-defined inserters and extractors be in line with common iostreams practice.[2]

Gathering the result of the above considerations we now know that our `date` inserter and extractor must have the following signatures:

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

and

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

Equipped with this knowledge, let us now implement a first version of the two operators.

The Simple Approach

The easiest thing to do when building an inserter or extractor for a new type is to decompose the type into its parts and make use of existing input and output operations for the contained elements. For instance:

- Input and output of an array type would be implemented by iterating through the array and using the inserters and extractors of the array element type for reading and writing each single element.
- For a class type, input and output of the object would boil down to input and output of the object's data members and base classes.

In our case a `date` object is decomposed into the day, month, and year contained in the `struct tm` data member of the `date` object. Each such element is an integral value and is inserted or extracted by means of the standard shift operator for type `int`:

The extractor:

```
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;
}
```

The inserter:

[2] The only sensible exception from that rule are inserters and extractors for user-defined stream classes, if the i/o operations depend on information that is specific to the new stream class.

```
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;
}
```

Note, that it usually is a good idea to make insertion and extraction complementary operations: An item should be written in a format that is understood by the input operation, so that you can read what you've written.

The `date` class still needs a minor modification: Both operations access private data members of class `date`, and must therefore be declared as friend functions to class `date`. Here's a completed version of class `date`:

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

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);
};
```

Refinements

The technique of creating new inserters and extractors by composing existing inserters and extractors is simple and powerful, but can be refined in several ways: more elaborated format control can be desired, errors might occur and must be handled, the format might depend on cultural conventions.

The predefined inserters and extractors in IOStreams follow a number of conventions in that they interpret format control parameters consistently, report errors in a uniform way, factor out culture-sensitive information into locales and facets, and so on. User-defined i/o operations should base their refinements on the same rules. We are going to discuss the areas of refinement together with the implementation techniques used for standard inserters and extractors. As an example we will then implement new date inserter and extractor which are reworked in all these areas in our next column.

Format Control

Our simple inserter from the example above has a tiny 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 discussion 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. Not all format flags are relevant to input and output of all types of objects. Some format flags apply to insertion and extraction of certain data types only. They are often irrelevant to

input and output of user-defined types. An example are the `oct`, `dec`, and `hex` format flags: they have an impact solely on input and output of integral values and can be ignored for formatting and parsing of dates, as in our example. Other format flags, such as `unitbuf`, `skipws` or the field width, are independent of the type of object inserted or extracted. They have an impact on user-defined types, too.

As a rule of thumb, you should first determine all format flags that you want to be relevant to the user-defined type that you intend to parse and format. Then understand how these flgs are used in the predefined inserters and extractors, and make sure that your user-defined operations interpret and manipulate them in exactly the same way.

An interesting format control element, that we've already mentioned, is the field width, because it is the only one that is not permanent, but is reset to 0 each time it was used. Stick to this rule if you decide to adjust output fields according to the field width, and reset the field width to 0 at the end of your inserter. Note also, that typical candidates of relevance for your user-defined inserter or extractor such as `unitbuf` and `skipws` are automatically performed as so-called prefix and suffix operations.

Prefix and Suffix Operations

It's a convention for stream input and output operations to carry out certain tasks prior (prefix activities) and subsequent (suffix activities) to any actual input or output. The prefix activities include flushing of a tied stream and skipping of white spaces. The suffix activities include flushing the stream if the `unitbuf` flag is set.
In the date inserter and extractor from the previous section, there is a certain performance overhead due to the use of existing inserters and extractors. The flushing, for instance, is performed for each single shift operation although it would only be necessary for input or output of the entire date. If we want to eliminate the overhead, we have to care about flushing ourselves.
In IOStreams, the prefix and suffix activities are encapsulated into classes called `sentry`, nested into the stream class templates `basic_istream` and `basic_ostream`. The constructors of these classes perform the prefix activities; the destructors carry out the suffix activities. The `sentry` classes have an `operator bool()`, which allows check for success of construction (i.e. success of the prefix operations).
The standard permits that the provider of an IOStreams library adds operations, beyond the ones listed above, to the constructors and destructors of the `sentry` classes.[3] Instead of using sentries you could manually add the prefix and suffix activities into your shift operators. Such explicitly implemented prefix and suffix activities, however, introduce potential portability problems, because the hidden vendor-specific additions would be missing. If you implement an inserter or extractor you should use the `sentry` classes in order to make sure that all necessary prefix and suffix tasks are carried out. When you compose existing shift operators, you implicitly do so anyway, because the predefined inserters and extractors use sentries. If you build shift operators on top of low-level i/o operations you have to care about prefix and suffix activities yourself.

Here is how you should use the `sentry` classes in you inserters and extractors:
(1) Create a local sentry object on the stack prior to any other activity in your shift operator.
The `sentry` constructor will be the first operation executed in your shift operator and performs all necessary prefix tasks: The local `sentry` object goes out of scope when the shift operator returns, its destructor will always, even in the presence of exceptions, be called and cares about the suffix tasks.
(2) Check for success of the prefix operations after construction of the sentry object by means of its `bool` operator.
The operator returns false in case of `!good()`, which means that either an error occurred or the end of the input was encountered. It's another convention that stream operations stop if the stream state is not good. Following this policy, one should return from the function if the check after construction of the sentry object does not indicate success.

This column does not leave us enough room to demonstrate the correct use of sentries, but we will make up for this omission in the next installment.

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. Again, there are rules to follow.

[3] Conceivable additions could be locking and unlocking of mutually exclusive locks for ensuring thread-safety of the IOStreams library, or maintenance of internal caches, etc.

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 exception mask. The predefined i/o operations for built-in and library types demonstrate the principle. Here is what a shift operator in IOStreams is expected to do in case of errors:

The stream state:

- Each stream has a state consisting of flags which indicate certain types of errors.
- The state flags have the following meaning: `badbit` indicates an error situation that is likely to be unrecoverable, whereas `failbit` indicates a situation that might allow you to retry the failed operation. The flag `eofbit` simply indicates the end of the input sequence.
- By convention, most IOStreams operations stop and have no further effect once one of the flags is set.
-
- *The exception mask:*
- By default none of the stream operations throws an exception. The user has to explicitly declare that he wants a stream to throw exceptions.
- For this purpose every stream contains an *exception mask*, which consist of several exception flags. Each flag in this mask corresponds to one of the stream state flags `failbit`, `badbit`, and `eofbit`. For example, once the `badbit` flag is set in the exception mask, an exception will be thrown each time the `badbit` flag gets set in the stream state.
-
- *The output variable of an extractor:*
- The value of the output variable must be a valid value; it need not be meaningful in case of a failed extraction. Other than that there are no requirements imposed by IOStreams. It is, however, common sense that the behavior should be consistent.

Inserters and extractors for user-defined types should obey these policies for error indication in IOStreams. Here is what you need to do when implementing input and output operations:

(1) *Detect error situations.* Problems can be reported from any component you invoke for accomplishing the task of formatted input and output. Examples are a `bad_alloc` exception thrown by an invoked operation due to memory shortage, or failure of an invoked stream operation indicated by a bit set in the stream state. Also, the internal logic of your inserter or extractor can identify error situations, e.g. extraction of incomplete or invalid dates in our example. Therefore:

- Catch all exceptions.
- Check all return codes, the stream state, or other error indications.
- Check the consistency of extracted or inserted data, if possible and necessary .

(2) *Indicate error situations*. Have your inserter and extractor report errors according to the IOStreams principles, that is:

(3) Set the stream state flags as follows:

- `ios_base::badbit` to indicate loss of integrity of the stream. Typical problems of that category are:
  - Memory shortage.
  - Errors reported from stream buffer functions.
- `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. It is important that you rethrow the original exception rather than raising an IOStreams exception of type `ios_base::failure`. The original exception is likely to convey information that precisely describes the error situation, whereas `ios_base::failure` is a rather non-specific, general iostreams error. If you do not retain the original exception, useful information might get lost.

(3) *Care about the output variable of an extractor.* Find a concept for the value you return in the output variable of a failed extractor. Ideally, one would retain or restore the original value of the output variable, so that the variable is unaltered in case of a failure. Another strategy is clearing the content

and returning a nil value in case that no valid value could be extracted. This is only doable if there is a nil value for the respective type. Examples of nil values are a zero-length string or a date like 00-00-00.

Again, this column is too short for showing a concrete use of these techniques; we will refine the `date` example in our next column and will then return to the error indication issue and show how the techniques described above can implemented.

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. In fact, they use a format that does not match any recognized date formatting rule we know of. We want to change this and intend to internationalize our date inserter and extractor. Users of our shift operators shall be allowed to indicate and switch cultural environments, and as a result the formatting and parsing shall be adjusted accordingly. In IOStreams, internationalized i/o operations are implemented by factoring out culture-sensitive parsing and formatting into exchangeable components: locales and facets. Here is a short recap of locales and facets:

The purpose of a locale is to represent a cultural area and contain all the culture-dependent information and services relevant for that area. They are used by components that deal with culture-sensitive data for retrieving information about cultural differences. Inserters and extractors in IOStreams are examples of such components: they need to know how culture-sensitive items like numeric values and dates are formatted.

The services and information in a locale are organized in smaller units, the facets. Often, there is not just a single facet for a certain problem, but a group of facet types is related to one problem domain. For instance, the standard facets `num_put`, `num_get`, and `numpunct` together represent the knowledge about numeric formats. Each locale object represents a particular cultural area and contains facets for that culture. A German locale, for example, has the numeric facets for German numeric format, and a US locale has the numeric facets for US formats.

Locales are provided to IOStreams on a per stream basis. That is, each stream has a locale of its own. The locale attached to a stream can be replaced. Locale-sensitive i/o operations are expected to use a stream's current locale object in order to achieve adaptability to cultural environments.

If you implement inserters and extractors for culture-sensitive data types you should take the following steps:
(1) Identify the culture dependencies that are related to the respective data type.
(2) Encapsulate relevant culture-dependent rules and services into facets.
This requires implementation of new facets types, if the necessary functionality is not yet available in a locale.
(3) Use the stream's locale and its facets for implementing your internationalized inserters and extractors.
Users of your culture-sensitive inserters and extractors must first create locales that contain the necessary facets and then imbue streams with such locales, by which the necessary facets are made available to your inserters and extractors.

Provide locales that contain the necessary facets, or provide means for creating such locales.
Every locale contains at least all standard facets. If you need non-standard facets you must support creation of these locales.

In our example we can use existing standard facets (namely `time_put` and `time_get`) for the insertion and extraction of `date` values and for this reason we do not have to deal with the creation of user-defined facets. A discussion of user-defined facets can be found in /2/.

I/O Operations

The inserter and extractor from the above example were built on top of existing shift operators. Composition of existing shift operators is not the only way of implementing new inserters and extractors. Alternatively, you can insert and extract items via stream iterators (see /1/ for more information about stream iterators). Stream iterators themselves are based on existing shift operators.

Hence, the use of stream iterators is another way of building new inserters and extractors on top of formatted i/o operations.

Approaches of a different quality are use of unformatted i/o operations such as the `read()` and `write()` member functions of a stream or direct access to the stream buffer via stream buffer iterators. Both solutions require explicit implementation of the entire parsing and formatting logic. The guidelines given above for format control, prefix and suffix operations, error indication, and internationalization help you to implement such shift operators correctly. Our refinement of the date inserter and extractor in the next column will be based on the use of stream buffer iterators.

Conclusion

We leave you after this discussion of possible refinements, because the refined implementation of the inserter and extractor will take more than one or two additional pages. We will present it together with the how's and why's in our next column. Stay tuned.

References

/1/ Stream Iterators
Klaus Kreft & Angelika Langer
C++ Report, May 1999

/2/ Extending the Locale Framework - User-Defined Facets
Klaus Kreft & Angelika Langer
C++ Report, February 1998