
15.11. Input/Output Operators for User-Defined Types

As mentioned earlier in this chapter, a major advantage of streams over the old I/O mechanism of C is the possibility that the stream mechanism can be extended to user-defined types. To do this, you must overload operators `<<` and `>>`. This is demonstrated using a class for fractions in the following subsection.

15.11.1. Implementing Output Operators

In an expression with the output operator, the left operand is a stream and the right operand is the object to be written:

stream `<<` *object*

According to language rules, this can be interpreted in two ways:

1. As *stream* `.operator<<(` *object* `)`
2. As `operator<<(` *stream* `,` *object* `)`

The first way is used for built-in types. For user-defined types, you have to use the second way because the stream classes are closed for extensions. All you have to do is implement global operator `<<` for your user-defined type. This is rather easy unless access to private members of the objects is necessary (which I cover later).

For example, to print an object of class `Fraction` with the format *numerator* `/` *denominator*, you can write the following function:

Click here to view code image

```
// io/frac1out.hpp

#include <iostream>

inline
std::ostream& operator << (std::ostream& strm, const Fraction& f)
{
    strm << f.numerator() << '/' << f.denominator();
    return strm;
}
```

The function writes the numerator and the denominator, separated by the character `'/'`, to the stream that is passed as the argument. The stream can be a file stream, a string stream, or some other stream. To support the chaining of write operations or the access to the stream's state in the same statement, the stream is returned by the function.

This simple form has two drawbacks:

1. Because `ostream` is used in the signature, the function applies only to streams with the character type `char`. If the function is intended only for use in Western Europe or in North America, this is no problem. On the other hand, a more general version requires only a little extra work, so it should at least be considered.
2. Another problem arises if a field width is set. In this case, the result is probably not what might be expected. The field width applies to the immediately following write; in this case, to the numerator. Thus, the statements

Click here to view code image

```
Fraction vat(19,100);    // I'm German and we have a uniform VAT of 19%
std::cout << "VAT: \"" << std::left << std::setw(8)
          << vat << '"' << std::endl;
```

result in this output:

```
VAT: "19      /100"
```

The next version solves both of these problems:

Click here to view code image

```
// io/frac2out.hpp

#include <iostream>
#include <sstream>

template <typename charT, typename traits>
inline
std::basic_ostream<charT,traits>&
```

```
operator << (std::basic_ostream<charT,traits>& strm,
             const Fraction& f)
{
    // string stream
    // - with same format
    // - without special field width
    std::basic_ostringstream<charT,traits> s;
    s.copyfmt(strm);
    s.width(0);

    // fill string stream
    s << f.numerator() << '/' << f.denominator();

    // print string stream
    strm << s.str();

    return strm;
}
```

The operator has become a function template that is parametrized to suit all kinds of streams. The problem with the field width is addressed by writing the fraction first to a string stream without setting any specific width. The constructed string is then sent to the stream passed as the argument. This results in the characters representing the fraction being written with only one write operation, to which the field width is applied.

As a result, the statements

```
Fraction vat(19,100);     // I'm German ...
std::cout << "VAT: \"" << std::left << std::setw(8)
          << vat << '"' << std::endl;
```

now produce the following output:

```
VAT: "19/100  "
```

Note that a user-defined overload of operator `<<` for types of namespace `std` will have limitations. The reason is that it is not found in situations using *ADL* (*argument-dependent lookup*, also known as *Koenig lookup*). This, for example, is the case when ostream iterators are used. For example:

Click here to view code image

```
template <typename T1, typename T2>
std::ostream& operator << (std::ostream& strm, const std::pair<T1,T2>&
p)
{
    return strm << "[" << p.first << "," << p.second << "]";
}

std::pair<int,long> p(42,77777);
std::cout << p << std::endl;     // OK

std::vector<std::pair<int,long>> v;
...
std::copy(v.begin(),v.end(),     // ERROR: doesn't compile:
          std::ostream_iterator<std::pair<int,long>>(std::cout,"\n"));
```

15.11.2. Implementing Input Operators

Input operators are implemented according to the same principle as output operators. However, input incurs the likely problem of read failures. Input functions normally need special handling of cases in which reading might fail.

When implementing a read function, you can choose between simple or flexible approaches. For example, the following function uses a simple approach, which reads a fraction without checking for error situations:

```
// io/frac1in.hpp

#include <iostream>

inline
std::istream& operator >> (std::istream& strm, Fraction& f)
{
    int n, d;

    strm >> n;          // read value of the numerator
    strm.ignore();      // skip '/'
    strm >> d;          // read value of the denominator

    f = Fraction(n,d);  // assign the whole fraction

    return strm;
}
```

The problem with this implementation is has that it can be used only for streams with the character type `char`. In addition, whether the character between the two numbers is indeed the character `'/'` is not checked.

Another problem arises when undefined values are read. When reading a zero for the denominator, the value of the read fraction is not well defined. This problem is detected in the constructor of the class `Fraction` that is invoked by the expression `Fraction(n,d)`. However, handling inside class `Fraction` means that a format error automatically results in an error handling of the class `Fraction`. Because it is common practice to record format errors in the stream, it might be better to set `ios_base::failbit` in this case.

Finally, the fraction passed by reference might be modified even if the read operation is not successful. This can happen, for example, when the read of the numerator succeeds, but the read of the denominator fails. This behavior contradicts common conventions established by the predefined input operators and thus is best avoided. A read operation should be successful or have no effect.

The following implementation is improved to avoid these problems. It is also more flexible because it is parametrized to be applicable to all stream types:

Click here to view code image

```
// io/frac2in.hpp

#include <iostream>

template <typename charT, typename traits>
inline
std::basic_istream<charT,traits>&
operator >> (std::basic_istream<charT,traits>& strm, Fraction& f)
{
    int n, d;

    // read value of numerator
    strm >> n;

    // if available
    // - read '/' and value of demonimator
    if (strm.peek() == '/') {
        strm.ignore();
        strm >> d;
    }
    else {
        d = 1;
    }

    // if denominator is zero
    // - set failbit as I/O format error
    if (d == 0) {
        strm.setstate(std::ios::failbit);
        return strm;
    }

    // if everything is fine so far
    // - change the value of the fraction
    if (strm) {
        f = Fraction(n,d);
    }

    return strm;
}
```

Here, the denominator is read only if the first number is followed by the character `'/'`; otherwise, a denominator of `1` is assumed, and the integer read is interpreted as the whole fraction. Hence, the denominator is optional.

This implementation also tests whether a denominator with value `0` was read. In this case, the `ios_base::failbit` is set, which might trigger a corresponding exception (see Section 15.4.4, page 762). Of course, the behavior can be implemented differently if the denominator is zero. For example, an exception could be thrown directly, or the check could be skipped so that the fraction is initialized with zero, which would throw the appropriate exception by class `Fraction`.

Finally, the state of the stream is checked, and the new value is assigned to the fraction only if no input error occurred. This final check should always be done to make sure that the value of an object is changed only if the read was successful.

Of course, it can be argued whether it is reasonable to read integers as fractions. In addition, there are other subtleties that may be improved. For example, the numerator must be followed by the character `'/'` without separating whitespaces. But the denominator may be preceded by arbitrary whitespaces because normally, these are skipped. This hints at the complexity involved in reading nontrivial data structures.

15.11.3. Input/Output Using Auxiliary Functions

If the implementation of an I/O operator requires access to the private data of an object, the standard operators should delegate the work to

auxiliary member functions. This technique also allows polymorphic read and write functions, which might look as follows:

Click here to view code image

```
class Fraction {
  ...
  public:
    virtual void printOn (std::ostream& strm) const;   // output
    virtual void scanFrom (std::istream& strm);        // input
    ...
};

std::ostream& operator << (std::ostream& strm, const Fraction& f)
{
    f.printOn (strm);
    return strm;
}

std::istream& operator >> (std::istream& strm, Fraction& f)
{
    f.scanFrom (strm);
    return strm;
}
```

A typical example is the direct access to the numerator and denominator of a fraction during input:

```
void Fraction::scanFrom (std::istream& strm)
{
   ...
   // assign values directly to the components
   num = n;
   denom = d;
}
```

If a class is not intended to be used as a base class, the I/O operators can be made `friend`s of the class. However, note that this approach reduces the possibilities significantly when inheritance is used. Friend functions cannot be virtual; as a result, the wrong function might be called. For example, if a reference to a base class refers to an object of a derived class and is used as an argument for the input operator, the operator for the base class is called. To avoid this problem, derived classes should not implement their own I/O operators. Thus, the implementation sketched previously is more general than the use of friend functions and should be used as a standard approach, although most examples use friend functions instead.

15.11.4. User-Defined Format Flags

When user-defined I/O operators are being written, it is often desirable to have formatting flags specific to these operators, probably set by using a corresponding manipulator. For example, it would be nice if the output operator for fractions, shown previously, could be configured to place spaces around the slash that separates numerator and denominator.

The stream objects support this by providing a mechanism to associate data with a stream. This mechanism can be used to associate corresponding data — for example, using a manipulator — and later retrieve the data. The class `ios_base` defines the two functions `iword()` and `pword()`, each taking an `int` argument as the index, to access a specific `long&` or `void*&`, respectively. The idea is that `iword()` and `pword()` access `long` or `void*` objects in an array of arbitrary size stored with a stream object. Formatting flags to be stored for a stream are then placed at the same index for all streams. The static member function `xalloc()` of the class `ios_base` is used to obtain an index that is not yet used for this purpose.

Initially, the objects accessed with `iword()` or `pword()` are set to `0`. This value can be used to represent the default formatting or to indicate that the corresponding data was not yet accessed. Here is an example:

Click here to view code image

```
// get index for new ostream data
static const int iword_index = std::ios_base::xalloc();

// define manipulator that sets this data
std::ostream& fraction_spaces (std::ostream& strm)
{
    strm.iword(iword_index) = true;
    return strm;
}

std::ostream& operator<< (std::ostream& strm, const Fraction& f)
{
    // query the ostream data
    // - if true, use spaces between numerator and denominator
    // - if false, use no spaces between numerator and denominator
    if (strm.iword(iword_index)) {
        strm << f.numerator() << " / " << f.denominator();
    }
```

```
    else {
        strm << f.numerator() << "/" << f.denominator();
    }
    return strm;
}
```

This example uses a simple approach to the implementation of the output operator because the main feature to be exposed is the use of the function `iword()`. The format flag is considered to be a Boolean value that defines whether spaces between numerator and denominator should be written. In the first line, the function `ios_base::xalloc()` obtains an index that can be used to store the format flag. The result of this call is stored in a constant because it is never modified. The function `fraction_spaces()` is a manipulator that sets the `int` value that is stored at the index `iword_index` in the integer array associated with the stream `strm` to `true`. The output operator retrieves that value and writes the fraction according the value stored. If the value is `false`, the default formatting using no spaces is used. Otherwise, spaces are placed around the slash.

When `iword()` and `pword()` are used, references to `long` or `void*` objects are returned. These references stay valid only until the next call of `iword()` or `pword()` for the corresponding stream object, or until the stream object is destroyed. Normally, the results from `iword()` and `pword()` should not be saved.[18] It is assumed that the access is fast, although it is not required that the data be represented by using an array.

[18] In general, returned pointers and references should not be saved when it is not clear that the lifetime of the object they refer to is long enough.

The function `copyfmt()` copies all format information (see Section 15.7.1, page 779), including the arrays accessed with `iword()` and `pword()`. This may pose a problem for the objects stored with a stream using `pword()`. For example, if a value is the address of an object, the address is copied instead of the object. If you copy only the address, it may happen that if the format of one stream is changed, the format of other streams would be affected. In addition, it may be desirable that an object associated with a stream using `pword()` be destroyed when the stream is destroyed. So, a deep copy rather than a shallow copy may be necessary for such an object.

A callback mechanism is defined by `ios_base` to support such behavior as making a deep copy if necessary or deleting an object when destroying a stream. The function `register_callback()` can be used to register a function that is called if certain operations are performed on the `ios_base` object. It is declared as follows:

Click here to view code image

```
namespace std {
  class ios_base {
    public:
      // kinds of callback events
      enum event { erase_event, imbue_event, copyfmt_event };
      // type of callbacks
      typedef void (*event_callback) (event e, ios_base& strm,
                                      int arg);
      // function to register callbacks
      void register_callback (event_callback cb, int arg);
      ...
  };
}
```

The function `register_callback()` takes a function pointer as the first argument and an `int` argument as the second. The `int` argument is passed as the third argument when a registered function is called and can, for example, be used to identify an index for `pword()` to signal which member of the array has to be processed. The argument `strm` that is passed to the callback function is the `ios_base` object that caused the call to the callback function. The argument `e` identifies the reason why the callback function was called. The reasons for calling the callback functions are listed in Table 15.39.

Table 15.39. Reasons for Callback Events

Event	Reason
`ios_base::imbue_event`	A locale is set with `imbue()`
`ios_base::erase_event`	The stream is destroyed or `copyfmt()` is used
`ios_base::copy_event`	`copyfmt()` is used

If `copyfmt()` is used, the callbacks are called twice for the object on which `copyfmt()` is called. First, before anything is copied, the callbacks are invoked with the argument `erase_event` to do all the cleanup necessary, such as deleting objects stored in the `pword()` array. The callbacks called are those registered for the object. After the format flags are copied, which includes the list of callbacks from the argument stream, the callbacks are called again, this time with the argument `copy_event`. This pass can, for example, be used to arrange for deep copying of objects stored in the `pword()` array. Note that the callbacks are also copied and the

original list of callbacks is removed. Thus, the callbacks invoked for the second pass are the callbacks just copied.

The callback mechanism is very primitive. It does not allow callback functions to be unregistered except by using `copyfmt()` with an argument that has no callbacks registered. Also, registering a callback function twice, even with the same argument, results in calling the callback function twice. It is, however, guaranteed that the callbacks are called in the opposite order of registration. Thus, a callback function registered from within another callback function is not called before the next time the callback functions are invoked.

15.11.5. Conventions for User-Defined Input/Output Operators

Several conventions that should be followed by the implementations of your own I/O operators have been presented. These conventions correspond to behavior that is typical for the predefined I/O operators. To summarize, these conventions are as follows:

- The output format should allow an input operator that can read the data without loss of information. Especially for strings, this is close to impossible because a problem with spaces arises. A space character in the string cannot be distinguished from a space character between two strings.
- The current formatting specification of the stream should be taken into account when doing I/O. This applies especially to the width for writing.
- If an error occurs, an appropriate state flag should be set.
- The objects should not be modified in case of an error. If multiple data is read, the data should first be stored in auxiliary objects before the value of the object passed to the read operator is set.
- Output should not be terminated with a newline character, mainly because it is otherwise impossible to write other objects on the same line.
- Even values that are too large should be read completely. After the read, a corresponding error flag should be set, and the value returned should be some meaningful value, such as the maximum value.
- If a format error is detected, no character should be read, if possible.