Research report 107

FORMATTED STREAMS

Extensible Formatted I/O for C++ Using Object-Oriented Programming

Mark Rafter

(RR107)

Abstract

In a language that allows the programmer to define new types, two characteristics of a formatted I/O system are desirable — it should be extensible and type-secure. We make a demand of a language when we require that such a formatted I/O system be expressible entirely within it.

In this paper we demonstrate that `C++` is expressive enough to meet this demand. An extension to the `C++ stream` I/O library is described that provides formatting capabilities in the style of the `C stdio` library. An example of its use is:

```
cout["log of %d is %7f"] << x << log(x)
```

An object-oriented implementation of this extension is described. The language features of `C++` that make this implementation possible are identified to encourage the use of this approach with other languages.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

October 1987

CONTENTS

FORMATTED STREAMS

Extensible Formatted I/O for C++ Using Object-Oriented Programming

Mark Rafter
Computer Science Department
Warwick University
England

ABSTRACT

In a language that allows the programmer to define new types, two characteristics of a formatted I/O system are desirable — it should be extensible and type-secure. We make a demand of a language when we require that such a formatted I/O system be expressible entirely within it.

In this paper we demonstrate that C++ is expressive enough to meet this demand. An extension to the C++ stream I/O library is described that provides formatting capabilities in the style of the C stdio library. An example of its use is:

```
cout["log of %d is %7f"] << x << log(x)
```

An object-oriented implementation of this extension is described. The language features of C++ that make this implementation possible are identified to encourage the use of this approach with other languages.

1. Introduction

Formatted I/O, the ability to specify the field width, accuracy, radix ... etc for a data item, is popular. A good formatted I/O system can provide the programmer with a compact notation well-suited to the task in hand (in effect a little language[1] for I/O applications). This partly explains the continued use of formatted I/O even when, as in the case of the C programming language[2], other aspects of the programming environment are quite hostile towards its use.

What support should a language provide for formatted I/O? The answer to this depends on the language in question. If the set of data-types provided by the language is fixed, then a built-in system of formatted I/O may often be adequate (e.g. in awk[3]). However, when the language allows the programmer to define new types, a built-in system of formatted I/O is inadequate, We require a formatted I/O system that is extensible.

How should these extensions be expressed? For a general-purpose language this is an awkward question. If the langauge is inadequate for the expression of its own formatted I/O system it can hardly claim success in being general-purpose. The situation becomes worse when we require that the formatted I/O system be type-secure.

The C programming language is equipped with a formatted I/O system in the form of the stdio library[2]. Although this library is implemented in C, it is neither extensible nor type-secure. This is due to the limited nature of C. The C++ programming language[4] performs better than C. It is expressive enough to implement an I/O system (the stream I/O library) that is both extensible and type-secure. However, the stream I/O library provides only crude capabilities for formatted I/O; it is basically an I/O system without format control.

In this paper we address the question: Is C++ expressive enough to allow its stream I/O library to be extended to provide format control facilities in the style of C's stdio library without compromising either extensibility or type-security? We show that C++ is sufficiently expressive. A possible extension of the stream I/O library that includes format control is described, and an object-oriented realisation of this in C++ is examined in detail. The features of C++ that provide essential support in this approach to formatted I/O systems are summarised. The formatted I/O system presented obeys criteria that we want to impose on all such systems. No use is made of complex programming conventions that are unenforceable at compile time. Extensibility is not achieved by allowing users to modify standard library sources.

This paper has several sections. The first establishes certain terminology, and describes both the C and the C++ I/O systems. An ability to read C will be useful, but no knowledge of C++ is assumed; points of fine detail relating to C++ have been relegated to an appendix, cross referenced using the notation *[APP0]*. The second section describes a formatted I/O extension to the C++ I/O library. The general form of the extensions is described, emphasis being placed on how we would like formatted I/O to work rather than on exactly how this is achieved. The details of the implementation are contained in the third and fourth sections; the third is concerned with how the implementation would be achieved in a completely object-oriented language, and the fourth with how to map this implementation into C++. The final section of the paper examines the degree to which our initial aims have been achieved, and summarises the features of C++ that provide essential support in this approach to formatted I/O systems.

2. The C and C++ I/O Systems

In a language that allows the programmer to define new types, two characteristics of formatted I/O are desirable — it should be *extensible* and *type-secure*. The meaning of extensibility is clear: it should be possible to input and output the user-defined types, and not only the base types of the language.

Type-security, however, is covered by a spectrum of reasonable interpretations. At one extreme, we may demand that a *successful* input operation for a variable of type `T` should always operate on a representation produced by `T`'s output operation. Under such a scheme, the output representation of an integer value could not be input as a floating point value. At the other end of the spectrum, we may merely demand that illegal values and operations are never generated by I/O operations. For example, under a scheme like this, the output representation of an integer value could be input as a floating point value; also an attempt to

output a character array as a though it were an integer would be handled gracefully. This should perhaps be called *type-robustness* rather than *type-security* but, to avoid terminological clashes with Stroustrup[4], we will use the latter term.

2.1 The C stdio I/O System

The C language `fscanf` and `fprintf` functions provide an example of a formatted I/O system that is *not at all* type-secure. The following C fragment reads an octal integer value into `x` and then prints the decimal representation of the value together with some explanatory text and the value's logarithm. This example is written in ANSI C[5], which differs from older versions of C[2] by allowing the types of external function arguments to be declared, as has been done for the `log` function in the example. Amongst other changes, ANSI C also provides limited implicit type conversion of function arguments, e.g. the integer argument `x` supplied to the `log` function is converted to a `double`, which the `log` function expects.

```
int     x;
extern  double  log( double );  /* declare log function */

fscanf(  stdin,  "%o", &x );
fprintf( stdout, "log of %d is %7f", x, log(x) );
```

The first argument of both `fscanf` and `fprintf` identifies a file. The second argument is a character array which is used as a format-specifier. Within the format specifier, the character `%` precedes the formatting information which relates to one of the later arguments. In this use of `fprintf`, `%d` means "print an integer decimal representation for the integer value `x`", and `%7f` means "print a floating point representation with a minimum field width of 7 characters for the double value `log(x)`". In this use of `fscanf`, `%o` means "read an octal integer representation for the integer variable `x`". Because in C all arguments are passed by value, `fscanf` is passed `&x`, a *pointer* to the variable `x`.

Both `fscanf` and `fprintf` take a varying number of arguments of varying types. At run-time, they deduce the number and types of their trailing arguments from the information in the format-specifier. Errors will result if either function is provided with trailing arguments that disagree in type or number from those implied by the format-specifier. Such errors often cause immediate program termination.

2.2 The C++ stream I/O System

The C++ stream I/O system addresses both extensibility and type-security, but lacks format control[4]. Stroustrup is aware of this lack, and provides some minimal support through the functions `hex`, `oct`, `dec` and `form`. These are unsatisfactory; `form`, in particular, provides all of the problems to be found with `fprintf`.

If we omit octal input and output field-width specification, the previous C example can approximated to using C++ stream I/O as follows

```
int     x;
extern  double  log( double );  /* declare log function */

cin  >> x;
cout << "log of " << x << " is " << log(x);
```

In this, `>>` is regarded as an input operator and `<<` as an output operator. `cin` is an input stream (type `istream`) and `cout` is an output stream (type `ostream`). Control appears to be passed to each data object in turn, and each data type seems to "know" how to output or input itself. We will examine how this works.

The infix notation

```
cin >> x
```

is equivalent to the functional notation

```
operator>>( cin, x )
```

where **operator>>** is the name of a user-defined operator function *[APP1]*. User-defined operators are genuine (overloaded) functions — they just allow two notations to be used for function call.

The above operator could be declared as

```
istream & operator>>( istream & strm, int & i );
```

In this, the first occurrence of **istream &** is the return type of the operator; this is dealt with below. The operator's left and right-hand operands are declared as the function arguments **istream & strm**, and **int & i**. The **&** in these declarations means *reference to*; the equivalent Pascal[6] declaration of **i** would be **var i:integer**. Reference parameters remove the need to explicitly work with pointers to variables. Thus, the argument **strm** is a reference to an input stream that will supply the data for the input operation, and **i** is a reference to the integer variable which is to receive a value.

For an operator function, and for any other overloaded function, the declared types of the function's formal arguments determine which function definition applies in a function call. Below is an example of a definition of an operator that outputs a C string (a zero-terminated character array). It is defined in terms of a similar operator that outputs single characters. *[APP2]*

```
ostream & operator<< ( ostream & out, char data[] )
{
        int i = 0;
        while( data[i] != 0 )
        {
                out << data[i];
                i = i+1;
        }
        return out;
}
```

User-defined operators have the usual syntactic properties that C associates with their tokens i.e. number of operands, associativity, and precedence. For instance, because the token **<<** is left-associative, the following

```
cout << "log of " << x
```

and

```
operator<<( operator<<( cout , "log of " ) , x )
```

are equivalent.

The convention "all I/O operators return a reference to the stream that they use" is adopted to allow I/O operators to be cascaded as above. This is why they have the return types **istream &** or **ostream &**, and why the statement

```
return out;
```

occurs in the definition of the output operator for strings above.

If we examine our example in the light of the above :

```
cin  >> x;
cout << "log of " << x << " is " << log(x);
```

we see that control flow in the stream I/O system is data-driven. The types of the operands of the overloaded I/O operators determine which functions are to be called, and the associativity of these operators is exploited to produce the required sequencing of I/O operations. To make this exploitation possible, the *convention* was adopted that I/O operators return a reference to the stream that they operate on. In general, the use of conventions indicates either bad style or a weakness in the language being used.

However, the use of this convention by the stream I/O system can be tolerated because it is simple, and (most importantly) the compiler can detect important transgressions at compile time.

The C++ stream I/O library lacks the formatting capabilities of stdio, but has important advantages over the C library. The I/O system is type-secure and user-extensible. Also, error-prone functions which take a varying number of arguments have been banished.

3. Formatted I/O for C++

The C++ stream I/O system can be made format-specifier-driven rather than data-driven. This can be done without losing type-security, extensibility, or the convenient operator based I/O idiom. In this section we will concentrate on the notation used for the formatting, its meaning in C++, and the control-flow mechanism that underlies the formatted I/O system. This mechanism presents a natural dilemma, which is resolved later as the fine detail of the implementation is presented.

When we incorporate **fprintf**-like format-specifiers into the stream I/O system, our example becomes

```
int     x;
extern  double  log( double );

cin["%o"] >> x;
cout["log of %d is %7f"] << x << log(x);
```

We want this notation to give the impression that a format-specifier is attached to a stream and maintains overall control as the I/O operations proceed. The format-specifier outputs its pieces of plain text, and at the format control points marked by **%**, it permits the data objects to input or output themselves. A data object gets any formatting parameters by "reading"this information from its format control point.

Again, we need to examine how the example works. The notation

```
cin["%o"]
```

is another example of a user-defined operator. Here the index operator, **[]**, is overloaded. It combines an input stream, **cin**, and a character array, **"%o"**, to yield a formatted input stream (type **fistream**). Similarly,

```
cout["log of %d is %7f"]
```

is another overloading of the index operator. Here it combines an output stream and a character array to yield a formatted output stream (type **fostream**).

As before we can re-write

```
cout["log of %d is %7f"] << x << log(x)
```

using functional notation. This time, for the sake of layout compactness, we have abbreviated the C++ keyword **operator** as **op**.

```
op<<( op<<( op[]( cout, "log of %d is %7f" ), x ), log(x) )
```

The only purpose of the index operators is to create a formatted stream object, **fstrm**, to control a formatted I/O operation. The operators bind access to two streams of data into **fstrm**.

- **fstrm.strm** is a reference to the unformatted stream upon which the formatted I/O is layered: i.e. the stream that the index operator was applied to.
- **fstrm.fmt** is a newly-created **istream** object that provides "read" access to parts of the format-specifier string. **fstrm.fmt** allows the data in the format-specifier string (supplied as the argument to the index operator) to be "read" using standard input operators. See section 4.3 for a discussion of this.

3.1 Control Flow

The introduction of the new types **fostream** and **fistream** allows the definition of formatted I/O operators that look like their unformatted counterparts but behave differently. The formatted I/O operators co-operate with the **fostream** objects created by the index operators to retain overall control throughout the formatted I/O operation.

After an index operator has created the formatted stream object, control passes from right to left through each of the formatted I/O operators. Each operator processes plain characters from the format-specifier until a format control point, marked by a **%** character, is encountered. For output operators, this processing consists of outputting the plain characters. In response to a format control point, the operator calls upon the next data object in the formatted I/O expression to perform its I/O operation. For this purpose, the object is provided with access to both **fstrm.strm** and **fstrm.fmt**. When the object's I/O operation is complete, it returns control to the formatted I/O operator and the processing of the remainder of the format-specifier takes place.

This control flow mechanism is quite flexible. It accommodates any housekeeping that needs to be performed before, or after, an object's I/O operation.

3.2 A Dilemma

To implement the above scheme, we must define the index operators that will form our formatted stream objects from a format-specifier string and an unformatted stream; this, in turn, requires us to define the formatted stream types. All of this is routine programming, and is omitted for the sake of brevity. The I/O operators, however, provide an interesting problem. The obvious implementation is to declare input and output operators in the spirit of the unformatted I/O system. To cope with our example

```
cout["log of %d is %7f"] << x << log(x);
```

we would declare an operator

```
fostream &
operator<<( fostream & fstrm, int x )
{
        house-keeping before object I/O;
        object-I/O for x;
        house-keeping after object I/O;
        return fstrm;
}
```

to handle the integer **x**, and a similar operator to handle the double **log(x)**. We have used some pseudo-code in order to suppress details of the house-keeping operations.

This approach presents us with a dilemma.

- We could make the provision of the formatted I/O operators the responsibility of data-type designers. Then, when a new data-type is designed, any associated formatted I/O operator will be required to conform to the model given by the operator above. This would be the adoption of a programming *convention.*

  Unfortunately, we have no way of enforcing such a convention. So, potentially serious errors, e.g. omitting one of the house-keeping steps, will only manifest themselves at run-time. This leads us to reject the provision of I/O operators by data-type designers on the grounds that this is unacceptably insecure.

- Alternatively, we may require that data-type designers only provide functions to perform the object-I/O. Then the provision of the I/O operators which will call the object-I/O functions becomes the responsibility of the formatted I/O system. This allows the implementor of the formatted I/O system to ensure that the programming convention is respected.

Unfortunately, this alternative is not possible if the formatted I/O system is to remain extensible while not allowing users to modify standard library sources. The implementor of the formatted I/O system is in no position to know in advance which data-types will be invented, and thus cannot provide the required operators. (Of course, this is the problem that functions like `fprintf` run up against. The solution taken there is to provide support only for the types that are known about in advance, i.e. the base types of the language.) We are forced to reject this alternative on the grounds of inextensibility.

We will see that the inextensibility of the second alternative can be overcome by using object-oriented programming.

4. An Object Oriented Implementation

In the previous sections we have loosely used expressions such as "the data objects input or output themselves". We must now clarify this terminology.

When programming with user-defined data-types, it is common to refer to variables as *objects* on which operations are defined. When the programming language being used allows data-type encapsulation, this nomenclature becomes the norm. Consequently, programming with objects of encapsulated data types is sometimes referred to as object-oriented programming (e.g. [7] chapter 17). In other sources[8] [9], and in this paper, the term object-oriented programming (OOP) is reserved for the application of specific techniques over and above those of data encapsulation.

We say that a language supports OOP when two features are present:

- The ability to manipulate objects of disparate types uniformly.

  Type hierarchies are one such mechanism. In a type hierarchy, general types include more specialised types as sub-types. Uniform manipulation can then be achieved by "forgetting" the details of an object's type, and manipulating it merely as an object of a (less specialised) super-type.

  A completely object-oriented language has a most general type, `obj`, which includes all other types as special cases. An example of such a language is Smalltalk[8], where the most general type is the class `Object`.

  Languages may have type hierarchies without being competely object-oriented — in these it is not possible to uniformly manipulate all objects in terms of some most general type. Examples are: Objective C[9], SIMULA[10] and C++[4].

- The ability to define operations which may be applied to objects uniformly (i.e without exact knowledge of an object's type) but which act on objects non-uniformly (i.e in a way dependent on the object's type).

  The inheritance mechanism in type hierarchies can be strengthened to achieve this. The inheritance mechanism allows that an operation defined for a "general" data type may also be applied to any object of a sub-type. In this case the operation's definition is inherited unchanged by the sub-type. However, this inherited definition may be overridden by a redefinition in the sub-type. The strengthening of the inheritance mechanism allows that such a redefinition applies, even when an object of the sub-type is being (uniformly) manipulated as an object of its super-type. This implies the need for some sort of type-labelling of objects.

  Examples of this mechanism are: method invocation — Smalltalk and Objective C; virtual functions — SIMULA and C++.

OOP can be used to solve problems of extensibility. Type hierarchies may be extended by adding new sub-types without modifying the existing parts of the hierarchies. Virtual functions allow operations to be specified on these extensible hierarchies without demanding *a priori* knowledge of the full type hierarchies.

When we use expressions such as "the data object outputs itself" we mean that a call is made of the output function defined for the object. In some settings, the object will come from a type hierarchy and the output

function will be a virtual function. In this case, the correct definition of the function will be selected even if the object is manipulated as if it is a member of its super-type.

4.1 A Solution

The solution to the dilemma presented at the end of the last section takes a simple form in a completely object-oriented language. To avoid introducing another language, we will (for the rest of this section only) adopt the *convenient fiction* that C++ is completely object-oriented, having all types as sub-types of the type `obj`. The additional problems of how to map the solution into a language like C++ are dealt with later.

We present the solution in two parts using the output operator as an example (input is similar). First, a *single* formatted output operator is defined. This will output objects of the hypothetical type `obj`

```
fostream & operator<<( fostream & fstrm, obj & ob )
{
        house-keeping before object I/O;
        object-I/O for ob;
        house-keeping after object I/O;
        return fstrm;
}
```

Because of the assumption that all types are subtypes of `obj`, this operator can be applied to all types of objects. Our example

```
cout["log of %d is %7f"] << x << log(x);
```

involves two applications of this one operator; once with an integer, `x`, as its right-hand operand, and once with a double, `log(x)`. This is type-correct in both cases because the operands are objects of sub-types of the type `obj`, and hence they are both `obj` objects in their own right.

The second part of the solution is to perform the object I/O by using a virtual function, `print`, that can be redefined for each type in the system. We demand that all types in the system be equipped with a virtual `print` function. Object I/O in the output operator becomes:

```
ob.print( fstrm.strm, fstrm.fmt );
```

where: `ob` is the object to be output; `print` is a two-argument virtual function for the object `ob`; `fstrm.strm` is the unformatted stream to receive the representation of `ob`; `fstrm.fmt` is the way of accessing the appropriate part of the format-specifier.

4.2 The Example

We will examine some of the events that take place as our example:

```
cout["log of %d is %7f"] << x << log(x);
```

is executed.

1. First, the index operator creates a formatted output stream object from `cout` and the string

   ```
   "log of %d is %7f"
   ```

   The formatted output stream object is actually anonymous but we will call it `fstrm`, the name it is known by in the formatted output operator defined above. `cout` becomes the unformatted output stream, `fstrm.strm`, to be used by `fstrm`. The string is the data for the format-specifier, `fstrm.fmt`, of `fstrm`.

2. The index operator returns a reference to `fstrm`.

3. The formatted output operator is invoked for `fstrm` with `x` as its other argument.

4. The operator performs its initial house-keeping. This involves reading `"log of "` from the format specifier `fstrm.fmt` and printing this on `fstrm.strm`.

5. The formatted output operator arranges for the object I/O to be done by calling the virtual `print` function for `ob`.

```
ob.print( fstrm.strm, fstrm.fmt );
```

   This function is provided with both `fstrm.fmt` and `fstrm.strm` as arguments. `fstrm.fmt` allows the `print` function to read details of the format specifier, and `fstrm.strm` is somewhere for the function to write its output.

6. The virtual `print` function appropriate to the type of `ob` is determined. In this case `ob` is a reference to `x` so the `print` function for the integer `x` is selected.

7. The `print` function executes for the integer `x`. It determines the way in which `x` should be printed by reading data (a `character` 'd') from the format specifier `fstrm.fmt`.

8. The `print` function prints `x` in the way specified (decimal) on the unformatted stream `fstrm.strm` and then returns.

9. The formatted output operator performs its final house-keeping. It reads the string " `is` " from `fstrm.fmt`, prints it on `fstrm.strm`, and returns yielding a reference to `fstrm`. Access to the format-specifier is explained in section 4.3.

10. The process continues with the formatted output operator being called for `fstrm` with the value `log(x)` as the other argument. The sequence of events is the same as for the case of integer `x`, but now the virtual `print` function will be selected on the basis of the type (`double`) of the expression `log(x)`.

4.3 Access to the Format-Specifier

Access to the format-specifier by the `print` and `read` virtual functions needs to be provided in a controlled way. One approach to this is to equip objects of the types `fistream` and `fostream` with a procedural interface through which parts of the format-specifier may be "read". The `fistream` and `fostream` objects are then in a position to keep track of how much of the format specifier has been processed. They can implement quoting conventions (e.g. a plain `%` character is represented as `%%` for `fprintf`) or impose access restrictions (e.g. ensure that only the current control point of the format-specifier can be "read").

Rather than making an arbitrary choice about the form of this procedural interface, we choose to make the "read" operation a genuine stream I/O input operation. The details of how this is achieved are merely outlined here as they are not central to this paper. The interested reader is referred to Stroustrup's paper[11].

Access to data in the format-specifier string can be via a genuine input operator, because the stream I/O system decouples the use of data delivered by a stream from the method of its delivery. The types `istream` and `ostream` define the interface to which unformatted I/O operators work — behind this interface are hidden the details of the stream data-types. The unformatted I/O operators see stream objects as byte-streams and are insensitive to the true nature of the stream objects that they are dealing with. The stream I/O system comes equipped with a hierarchy of data-types that allows stream access to data kept in external files, simple character arrays in memory, and circular buffers. This hierarchy may be extended by the programmer to provide stream access to user-defined data-types. By decoupling data-use from data-delivery, it is possible for one set of operators to provide I/O on files, arrays and user-defined data-types.

The object `fstrm.fmt` is the `istream` interface to an object that provides stream access to the format specifier string. The access that is to be provided will be more complicated than (say) the access for a character array, but the principles are the same.

5. The OOP Solution in C++

We now turn to mapping our object-oriented solution into C++. In C++, there is no type `obj` from which all other types are derived. The first step is to address this problem.

We declare a *container* type `fobj` to approximate to the type `obj`. Then, for each type, `T`, that is to be equipped with formatted I/O we declare another container type, `fobj_T`. `fobj_T` is declared as a sub-type of `fobj`. This provides a hierarchy of container types. Each object of a type in this hierarchy will be used to "wrap up" an object taking part in formatted I/O.

The second step is to define the I/O operators. The formatted input and output operators are defined in terms of the type `fobj`, and so may operate on any object of a sub-type of `fobj`. As before, the operators provide the scaffolding, but the real work is done by `print` and `read` virtual functions. These are defined on the `fobj` type hierarchy, and redefined appropriately for each of the sub-types.

The objects of the subtypes in the `fobj` hierarchy are not of interest in themselves; they will merely have been used to "wrap up" the objects taking part in the formatted I/O operations. Hence, in each sub-type the virtual `print` and `read` functions will specify how to perform an I/O operation on the the "wrapped-up" object.

The third step is to attend to the "wrapping-up". Explicit "wrapping-up" is undesirable, and can be avoided if each container type is equipped with a suitable *constructor*. A constructor for a type `B` taking a single argument of type `A` not only specifies how a newly-created object of `B` is to be initialised from an object of type `A`, but also defines a user-defined conversion from `A` to `B`. Such conversions are implicitly applied, e.g. when matching a function's actual arguments with the declared types of its formal arguments.

We equip each type `fobj_T` with a constructor taking a single argument of type `T`. This will then allow a formatted I/O operand of type `T` to be implicitly converted to (i.e "wrapped-up" into) an object of type `fobj_T`. Because `fobj_T` is a sub-type of `fobj`, the converted object is an acceptable right-hand operand for the formatted I/O operators.

This use of constructors allows an object of type `T` to be supplied as the right-hand operand for the formatted I/O operators: the "wrapping-up" has been automated (but see *[APP3]*).

5.1 Some C++ Details

We now turn to the implementation in C++ of the above outline. The base of the container type hierarchy is the type `fobj`. The declaration of `fobj` contains no data, but it does contain the declarations of the two virtual functions that are to be defined for the whole `fobj` type hierarchy.

```
struct fobj
{
        virtual void print( ostream & strm, istream & fmt );
        virtual void read(  istream & strm, istream & fmt );
};
```

The declaration of the output operator is straight-forward (input is similar). Again, we use some pseudo-code in order to suppress details of the house-keeping operations:

```
fostream & operator<<( fostream & fstrm, fobj & ob )
{
        house-keeping before object I/O;
        ob.print( fstrm.strm, fstrm.fmt );
        house-keeping after object I/O;
        return fstrm;
}
```

These operators, taken together with the definitions of the index operators and the formatted stream types, complete the scaffolding of the formatted I/O system. What remains to be provided is an example of how a programmer would use this scaffolding to build user-supplied formatted I/O for a particular data-type.

5.2 Formatted Output of C Strings

As an example of the use of the scaffolding, we will see how to provide a variant of formatted output for C strings. Two formats are catered for; the first outputs a string argument with all its lower-case letters

translated to upper-case. This is requested by a **%u** format parameter in the format specifier. The second format outputs a string argument in the normal way; this is requested by the usual **%s** format parameter. Any other format parameter is an error, and will be silently treated as **%s**.

First let us introduce a simple name, **string**, for the C string data type **char ***.

```
typedef char * string;
```

A container type, **fobj_string**, is necessary

```
struct  fobj_string: fobj
{
        string  data;
        fobj_string( string s ){ data = s; }
        print( ostream & strm, istream & fmt );
};
```

This is an example of a C++ **class**; it has been declared using the keyword **struct** to allow us to avoid discussing the C++ class-member visiblity rules. This container type declares three members. The first member, **data**, is used to point to the string undergoing formatted I/O; it is initialised in the constructor. The constructor is the second member; in this example the body of the constructor is supplied as part of the type declaration. The third member is the **fobj** hierarchy **print** function, indicating that it will be redefined for the **fobj_string** class. The re-definition of **print** uses the library functions **isalpha** and **toupper** to map lower-case letters into upper-case ones. For readability, its body is provided separately from the class declaration.

```
void fobj_string::print( ostream & strm, istream & fmt )
{
        int i = 0;
        char mapcase;
        fmt >> mapcase;          /* get format parameter */

        while( data[i] != 0 )
        {
                char    ch = data[i];
                if( mapcase=='u' && isalpha( ch ) )
                {
                        ch = toupper( ch );
                }
                strm << ch;
                i = i+1;
        }
}
```

An example of the use of the above is

```
cout["+%s+%u+"] << "-hello-" << "-world-" ;
```

which produces the output

```
+-hello-+-WORLD-+
```

on the output stream **cout**.

6. Conclusions

In demonstrating that C++ is expressive enough to extend the stream I/O system to include format specification, we have developed an approach to structuring a formatted I/O system that could be applied to other languages. Whether this is easy, unattractive, or impossible will depend on the facilities provided to the programmer by the language in question.

In this section we will identify the features of C++ that are of importance to the formatted I/O system presented here. C++ also provides difficulties; these, and inherent limitations in our approach to formatted I/O, will be discussed. First we summarise what the formatted I/O system achieves in extensibility, type-security, error handling, and the avoidance of programming conventions.

6.1 Achievements

Extensibility As new data-types are introduced, the user is free to provide virtual **`print`** and **`read`** functions to perform formatted I/O. These functions will interpret the parameters that they read from the format-specifier in ways appropriate to their data-type. For example, a data-type **`complex`** might need to provide a **`%z`** format-specifier.

Type-Security The type of the object taking part in formatted I/O determines the correct operator function to call using the virtual function mechanism. Although the format-specifier appears to control the progress of the formatted I/O statement, the format-specifier only controls the points at which the virtual functions are called. It is the types of the data objects that really determine which functions are to be called.

Format Errors Because the formatting information in the format-specifier is decoupled from the data objects to which it refers, it is possible for the two to be incompatible. For example, an output format appropriate to a string may be paired with an integer variable. But just as input operators should cope with incorrect data in the external data stream, I/O operators can (and should) gracefully handle unexpected formatting requests. Problems do not arise as a result of I/O operators manipulating objects of the wrong type because, as noted above, the type of the object determines the function called.

The above should be contrasted with the situation that arises when either **`fprintf`** or **`fscanf`** are presented with incompatible formats and data-types. Both **`fprintf`** and **`fscanf`** use the format-specifier, not the type information, to select the (incorrect) formatting algorithm which then operates on the data object in a way inappropriate to its type. Graceful error-handling is not possible; a common consequence is immediate program termination.

Programming Conventions The control flow conventions used in the formatted I/O system are quite complex; provision is made for housekeeping to be done both before and after each format control point. Any formatted I/O system that required its users to conform to such conventions would be quite unacceptable. Programming errors that were transgressions of these complex conventions could not be detected at compile time. The result would a formatted I/O system that encouraged bug-prone programs.

The complex control-flow conventions are hidden from the user in the scaffolding of the formatted I/O system. To extend the formatted I/O system to cope with a new type the user must provide a container type in the **`fobj`** hierarchy. The control flow convention demanded of this type is merely the provision of a **`print`** or **`read`** function.

6.2 Language Features of C++

Of the language features of C++ that are important to our formatted I/O system, one should need no mention — strong typing. Not only is this the foundation for type-security in our system, but also several of the other heavily-used language features cannot exist in its absence. Most of the language features that we identify here underpin not only our formatted I/O system but also Stroustrup's original stream I/O system.

Overloading The ability to express related ideas in different contexts using a common notation relieves the programmer of the burden of many irrelevant details. The same notation is used to output an integer, a complex number, a tree, or a string; moreover this notation is independent of whether the the data source is a file, a string, or a concurrent process.

User-Defined Operators User-defined functions that may be called using infix notation provide an extremely compact notation that seems well suited to I/O. This notation could be done without (e.g. see the expanded form of operator expressions) but the resulting long expressions of nested function calls obscure the programmer's intention.

Object-Oriented Programming Support Type hierarchies and virtual functions have important semantic content. Whereas Overloading and User-Defined Operators could be dismissed as "semantic sugar", mere psychological props, this is not the case here. In the development of the formatted I/O system we saw that a natural dilemma was reached which was resolved by applying the more powerful techniques of object-oriented programming. Without OOP the development would have failed at this point. It should be clear that languages that may be claimed to support object-oriented programming, but in fact merely provide support for programming with encapsulated data-types, will cause difficulties here.

User-Defined Type Conversion The ability to extend the set of implicitly-applied type conversions by defining an appropriate constructor was used to automatically wrap up data objects and inject them into the `fobj` type hierarchy, ready to take part in formatted I/O. This is convenient, but could be done by the explicit use of a function. The use of such an injection function would be ugly, and would clutter the formatted I/O expressions, but this is not its main drawback. Formatted I/O expressions would then look different to unformatted I/O expressions, and we would lose some of the advantages that accrue from using a common notation to express related ideas in different contexts.

6.3 Difficulties

The formatted I/O system takes the form of a library; because of this, it can only detect format errors at run-time, not compile-time. This is true even when it is possible to deduce the presence of an error at compile time.

A difficulty that seems to have no satisfactory resolution in C++ is illustrated by the last example, formatted output of C strings. The name of the data-type, `char*`, is a problem when we attempt to declare a container type for it in the `f_obj` hierarchy. In the example, we worked around the problem by introducing the type-name `string`. Ideally, the container type should be named `fobj_char*` but this would be syntactically incorrect in the class declaration, and would mean something quite unintended in other contexts. This is because C++ gives us no way to express the intended form of the `f_obj` hierarchy. What is needed is a way of re-expressing `f_obj` as a function on types combining aspects of both generic types and type hierarchies.

6.4 Further Work

Layering the formatted I/O system on top of the existing stream I/O system is an interesting exercise, but it means that separate operators for formatted and unformatted I/O are required. This is undesirable from the point of view of the person writting the operators; also, it means that mixing formatted and unformatted I/O is problematical. More work is needed to develop an integrated system. There are no fundamental obstacles here, rather the difficulty lies in achieving an implementation that is acceptably efficient for the case of unformatted I/O.

6.5 Summary

The C and C++ I/O systems have been examined and the question posed: Is C++ expressive enough to allow its stream I/O library to be extended to provide format control facilities in the style of C's stdio library without compromising either extensibility or type-security? It has been shown that C++ is sufficiently expressive to do this. An implementation of formatted extension to the C++ stream I/O system based on principles of object-oriented programming (type-hierarchies and virtual functions) has been examined in detail, and a method of realising this in C++ has been given. We have discussed the C++ language features that are essential to our approach — the methods themselves may be applied to other languages that provide the necessary support.

6.6 Acknowledgements

I am indebted to Professor Mathai Joseph, Kay Dekker, Jeff Smith, Steve Hunt, Julia Dain and Russel Quinn for their valuable comments on drafts of this paper.

7. Appendix

This appendix covers those points of C++ that were, in the interests of the exposition, presented in a simplified form.

APP1 The stream I/O operators for the base types of C++ are actually defined as member functions of the types **ostream** and **istream** rather than as the plain operator functions presented in this paper.

APP2 The output of `cout << 'a';` is the decimal string **97** on an ASCII system, rather than the expected letter **a**. This is because in C++, for the sake of compatibility with C, character constants are taken to have type **int** rather than type **char**. Thus, the character **'a'** in the above operator expression is actually an *integer*, not the character it appears to be. The situation is slightly better for the output of character *variables*. For these a reference-to-character output operator can be added to the type **istream**. A reference-to-character is distinct from a reference-to-integer, so the correct definition of the output operator is found by the C++ overloading rules. With this done, `cout << data[i]` outputs characters, not integers.

APP3 What appears to be a compiler bug prevents the constructor-based scheme for wrapping-up data objects from working. The appropriate object in the **fobj** hierarchy can be created automatically; and such an object can be automatically cast to its public base class. Unfortunately the release 1.1 C++ compiler from Bell Labs will not do both of these things automatically; although a careful reading of the language definition[4] indicates that it should.

The problem can be circumvented by the introduction of an output operator for each of the types in the **fobj** hierarchy. An operator to cope with the formatted output of the type **T** would be:

```
fostream & operator<<( fostream & fstrm, T & t )
{
        return fstrm << fobj_T( t );
}
```

This operator just explicitly states one of the conversions, that from type **T** to type **fobj_T**; the release 1.1 C++ compiler is then happy to do the other automatically.

The need to provide these unnecessary operators is ugly, but should not be viewed as part of the architecture of the formatted I/O system. However, we are lucky in that the task is mechanical, and does not introduce any unacceptable programming conventions.

References

1. Jon Bentley "Little Languages — Programming Pearls Column", *Comm ACM*, **vol29 No8**, 711-721, (Oct 1986).

2. B.Kernighan and D.Ritchie, *The C Programming Language*, Prentice Hall, New Jersey, 1978.

3. A.Aho, B.Kernighan and P.Weinberger, "Awk — A Pattern Scanning and Processing Language Programmer's Manual", *Computing Science Technical Report no. 118*, AT&T Bell Laboratories, New Jersey, June 1985.

4. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, New Jersey, 1986.

5. *Draft Proposed American National Standard for Information Systems — Programming Langauge C*, X3J11/86-017.

6. K.Jensen and N.Wirth, *PASCAL User Manual and Report*, edn 2, Springer-Verlag, New York, 1978.

7. G.Ford and R.Wiener, *Modula-2 A Software Development Approach*, John Wiley & Sons, New York, 1985.

8. A.Goldberg and D.Robson, *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, Massachusetts, 1983.

9. Brad Cox, *Object-Oriented Programming — an Evolutionary Approach*, Addison-Wesley, Massachusetts, 1986.

10. G.Birtwistle, O.Dahl, B.Myhrhaug, K.Nygaard, *SIMULA BEGIN*, Petrocelli/Charter, New York, 1975.

11. Bjarne Stroustrup, "An Extensible I/O Facility for C++", *1985 Summer Usenix Technical Conference Proceedings*, 57-70, Usenix Association, El Cerrito, 1985