An Extensible I/O Facility for C++

Bjarne Stroustrup

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill NJ 07974

ABSTRACT

This paper describes the classes `ostream` and `istream` designed to replace the `printf/scanf` family of input and output functions in C++. The operator `<<` is overloaded for ostreams to provide an single type-secure paradigm for output of both built-in and user-defined types. This paradigm typically yields output statements as short as or shorter than `printf()`. The operator `>>` handles input in a similar fashion. Conditions like reaching the end of a file or the corruption of a stream are handled by associating a state with each stream, rather than by returning "illegal values" like `EOF`. The C++ facilities used to implement stream i/o are briefly explained. These include classes providing data hiding, constructors providing initialization, destructors providing cleanup (in particular, flushing of buffers), operator overloading, and virtual functions providing uniform use of buffers with different strategies for handling underflow and overflow.

Introduction

Except for minor details C++ is a superset of the C programming language. In addition to the features of C, C++ provides Simula67-like classes, operator overloading, and a host of minor improvements. The parts of the C++ language that are used to specify the stream I/O facility are briefly explained. References 6-8 describe C++ in the detail necessary for writing programs in it.

The `printf()` family of functions provides simple, flexible, and terse formatted output[5] for C programs (and therefore also for C++ programs). However, uses of `printf()` cannot in general be type checked, and there is no convenient way for a user to deal with user-defined types in the same way as built-in types. Consider:

```
printf(stderr,"x = %s\n",x);
```

This is of course an error under all circumstances since only `fprintf()` and not `printf()` takes a `FILE*` like `stderr` as an argument. This problem is trivially handled in C++ where `<stdio.h>` declares

```
extern int fprintf(FILE*, char* ...);
extern int printf(char* ...);
```

thus catching that error at compile time. The ellipsis indicates that any number of arguments of any type may follow the initial arguments. However, had `x` been an `int` (rather than the `char*` expected by the `%s` in the format string) no error would have been detected until the program started printing garbage. Furthermore, had `x` been a user-defined type like `complex` there would have been no way of specifying the output format of `x` in the convenient way used for types "known to `printf()`" (for example, `%s` and `%d`). The programmer would typically have defined a separate function for printing complex numbers and then written something like this:

```
fprintf(stderr,"x = ");
put_complex(stderr,x);
fprintf(stderr,"\n");
```

This is inelegant, and would be a major annoyance in C++ programs where a non-trivial program typically uses several user-defined types for the manipulation of entities that are interesting/critical to an application.

Type-security and uniform treatment can be achieved by using a single overloaded function name for a set of output functions. For example:

```
put(stderr,"x = ");
put(stderr,x);
put(stderr,"\n");
```

The type of the argument determines which "put function" will be invoked for each argument. However, this is too verbose. The C++ solution using an output stream for which << has been defined as a "put to" operator looks like this:

```
cerr << "x = " << x << "\n";
```

where cerr is the standard error output stream (equivalent to the C stderr). So, if x is an int with the value 123, this statement would print

```
x = 123
```

followed by a newline onto the standard error output stream.

This style can be used as long as x is of a type for which operator << is defined, and a user can trivially define operator << for a new type. So, if x is of the user-defined type complex with the value (1,2.4), the statement above will print

```
x = (1,2.4)
```

on cerr.

The stream I/O facility is implemented exclusively using language features available to every C++ programmer. Like C, C++ does not have any I/O facilities built into the language. The stream I/O facility is provided in a library and contain no "extra-linguistic magic".

Output of built-in types

For output the class ostream is defined. The operator << ("put to") is defined to handle output of the built-in types:

```
class ostream {
        // ...
public:
        ostream& operator<<(char*);       // write
        ostream& operator<<(long);        // beware: << 'a' writes 97
        ostream& operator<<(double);

        ostream& put(char);               // put('a') writes a
        ostream& flush();

        // ...
                ostream(streambuf* s);         // bind to stream buffer
                ostream(int fd);               // bind for file
                ostream(int size, char* p);    // bind to vector
                ~ostream();
};
```

This class declaration defines the new type ostream. A class declaration is very much like a C struct declaration, except that a C++ class can have function members. Furthermore, a member of a class appearing before the public: label can only be used by the functions mentioned in the class declaration and is inaccessible to all other functions in the program. The internal representation of an ostream has been rendered inaccessible to a user in this way, and since it is not particularly interesting it has been omitted to simplify the discussion. The comment

```
// ...
```

is used to indicate this. In C++ // starts a comment that terminates at the end of line. Traditional C /* */ comments can also be used.

The operations <<, put() and flush() may be applied to an ostream. For example:

```
cerr << "x = ";
```

where cerr is an ostream will be interpreted as

```
cerr.operator<<("x = ");
```

An operator<< function returns a reference to the ostream it was called for so that another ostream can be applied to it. For example:

```
cerr << "x = " << x;
```

where x is an int, will be interpreted as

```
(cerr.operator<<("x = ")).operator<<(x);
```

In particular, this implies that when several items are printed by a single output statement they will be printed in the expected order: left to right. Since "x = " is a string the first operator<< function will be chosen to write it, and similarly the appropriate operator<< function will be chosen for x depending on x's type. Since x was an int it will be implicitly converted to a long (as if in an assignment) and passed to the second operator<< function. Floating point numbers are handled by the third operator<< function.

This facility for overloading function names and operators and then choosing the correct version to use for a particular call based on the types of the arguments is general in C++ and has nothing particular to do with I/O. Overloading enables the programmer to reduce the number of function names needed in a program by allowing several functions performing similar operations on objects of different types to share a name. In this particular case we avoid the printf(), fprintf(), and sprintf() name proliferation. The implicit type conversions reduce the number of functions needed. For example, there is a single function for handling the integral types: char, short, int, and long. There is no facility for printing unsigned values in a different way from signed values since the facility for resolving calls to overloaded functions in C++ cannot distinguish signed and unsigned types. Separate functions can, when needed, be used to handle such cases.

Some design details

It was necessary to define an output operator to avoid the verbosity that would have resulted from using an output function. But why <<? In C++, it is not possible to define a new lexical token, so one could not simply invent a new operator†.

The assignment operator was a candidate for both input and output, but it binds the wrong way. That is, cout=a=b would be interpreted as cout=(a=b), and most people seemed to prefer the input operator to be different from the output operator.

† Part of the reason for this "restriction" is that most "obvious" choices of new operators would create ambiguities and/or render legal C++ programs illegal. Consider, for example, these "possible operators": ->, **, <-, and //.

The operators < and > were tried, but the meanings "less than" and "greater than" were so firmly implanted in people's minds that the new I/O statements were for all practical purposes unreadable (this does not appear to be the case for << and >>). Apart from that, ´<´ is just above ´,´ on most keyboards and people were writing expressions like this:

```
cout << x , y , z;
```

It is not easy to give good error messages for this.

Another problem was that there are no character valued expressions in C++ (exactly like in C). In particular, ´\n´ is an integer (with the value 10 when the ASCII character set is used), so that

```
cout << "x = " << x << ´\n´;
```

writes the number 10 after x and not the expected newline. This and similar problems can be alleviated by defining a few macros

```
#define sp << " "
#define ht << "\t"
#define nl << "\n"
```

The example can now be written like this:

```
cout << "x = " << x nl;
```

Using non-syntactic macros is considered bad style in some quarters, but I like these (despite disliking macros in general).

Consider also these examples

```
cout << x << " " << y << " " << z << "\n";
cout << "x = " << x << ", y = " << y << "\n";
```

Most people find them hard to read because of the high number of quotes and because the output operator is visually too imposing. The macros above plus a bit of indentation can help here:

```
cout << x sp << y sp << z nl;
cout << "x = " << x
     << ", y = " << y nl;
```

Initialization and closing of output streams

There are two standard output streams cout and cerr corresponding to stdout and stderr. Naturally, the implementation involves a buffer that is occasionally flushed onto the associated output device. Flushing can be done explicitly like this:

```
cout.flush();
```

This is not required; the buffer is internal and hidden and will be flushed automatically when appropriate.

Two related questions comes to mind: How did a stream like cout get initialized, and how does it get flushed when the program terminates? Consider this program

```
#include <stream.h>

main()
{
        cout << "Hello, world";
}
```

The include directive ensures that the declarations needed to use an ostream are available. When class ostream was declared it was provided with constructors and a destructor. A constructor is a function that must be called whenever an object of its type is created. It is distinguished by the compiler by having the same name as its class. In particular, for an ostream the constructors

```
ostream(streambuf* s);          // bind to stream buffer
ostream(int fd);                // bind for file
ostream(int size, char* p);     // bind to vector
```

were declared so one of them must be called when an ostream is created. Which one to call will, as usual, be determined by the type of the arguments. The standard output streams are declared like this (using file bufferes as described below):

```
char cout_buf[BUFSIZE];
filebuf cout_file = filebuf(1,cout_buf,BUFSIZE); // UNIX output stream 1
ostream cout = ostream(&cout_file);

char cerr_buf[1];
filebuf cerr_file = filebuf(2,cerr_buf,0);       // UNIX output stream 2
                                                 // 0-length => unbuffered
ostream cerr = ostream(&cerr_file);
```

This code appears in the source for the stream I/O part of the C++ standard library, not in the ⟨stream.h⟩ header file. The C++ compiler/linker/loader is smart enough to figure out that the ostream constructor needs to be called for cout and cerr before main() is executed. Every static object of a class with a constructor in a program is handled in this way; this is not a "special feature" for I/O.

A destructor is a function that must be executed when an object of its type is destroyed (for example, when an object goes out of scope). The name of the destructor for a class is the complement operator ~ followed by the name of the class, for example

```
~ostream();
```

A destructor often complements a constructor by cleaning up a data structure initialized by the constructor. For example, the destructor for ostream flushes the buffer:

```
ostream.~ostream()
{
        flush();
}
```

In the case of a static object like cout the destructor is called after the execution of main().

The C standard I/O structures stdout and stderr also contain buffers that need flushing. This is done by "magic" in the code that execute a C program since C does not have a general feature for allowing a user to associate initialization and finalization code with data objects. Early C++ implementations could not cope with destructors for static objects and resorted to similar non-general "magic" to implement output streams.

Output of user-defined types

Consider a user-defined type:

```
class complex {
        double  re, im;
public:
        complex(double r = 0, double i = 0) { re=r; im=i; }

        friend  double  real(complex& a) { return a.re; }
        friend  double  imag(complex& a) { return a.im; }

        friend  complex operator+(complex, complex);
        friend  complex operator-(complex, complex);
        friend  complex operator*(complex, complex);
        friend  complex operator/(complex, complex);
        // ...
};
```

Operator << can be defined for the new type complex like this

```
ostream& operator<<(ostream&s , complex z)
{
        return s << "(" << real(z) << "," << imag(z) << ")";
};
```

and used exactly like a built-in type:

```
complex x(1,2);
// ...
cout << "x = " << x << "\n";
```

Note that defining an output operation for a user-defined type does not require modification of the declaration of class ostream, or access to the (hidden) data structure maintained by it. The former is fortunate since the declaration of class ostream resides among the standard header files to which the general user does not have write access. The latter is also important since it provides a good protection against accidental corruption of that data structure. It also makes it possible to change the implementation of an ostream without affecting user programs (see the acknowledgements).

Formatted output

So far << has been used only for unformatted output, and that has indeed been its major use in real programs. There are, however, a few formatting routines that create a string representation of their argument for use as output. Their (optional) second argument specifies the number of character positions to be used.

```
char* oct(long, int =0);          // octal representation
char* dec(long, int =0);          // decimal representation
char* hex(long, int =0);          // hexadecimal representation

char* chr(int, int =0);           // character, chr(0) is the empty string ""
char* str(char*, int =0);         // string
```

Truncation or padding will be done unless a zero-sized field is specified; then (exactly) as many characters as needed is used. For example:

```
cout << "oct(" << oct(x,6) << ") = hex(" << hex(x,4) << ")\n";
```

One can also used a printf style format string:

```
char* form(char* ...);            // printf format
```

Using form() one gets exactly the facilities and problems well known from use of printf(); it is actually sprintf() in disguise. Work is needed to get a satisfactory facility for providing formatted output of user-defined types without the elaboration and type-insecurities associated with the printf approach. In particular, it is probably necessary to find a standard way of providing the output function for a user defined type with information allowing it to determine space limitations, expectations about padding, left or right adjustment, etc., as expressed by its caller. A practical, but not ideal approach, is to provide functions for user defined types that, like the formatting functions above, produce a suitable string representation of the object for which they are called. For example:

```
class complex {
        float re,im;
public:
        // ...
        char* string(char* format) { return form(format,re,im); }
};
// ...
cout << z.string("(%.3f,%.3f)");
```

Input of built-in types

Input can be done using an `istream` defined analogous to an `ostream`. Here is a small complete program that reads in a number using the operator `>>` ("get from") on the standard input stream `cin`. The number read is assumed to be a number of inches and the program prints the equivalent number of centimeters:

```
#include <stream.h>

main()
{
        int inch;
        cout << "inches=";
        cin  >> inch;
        cout << inch << " in = " << inch*2.54 << " cm\n";
}
```

This can be compiled and run given the input `10` like this:

```
$ CC inch.c
$ a.out
inches=10
10 in = 25.4 cm
$
```

Note that the address-of operator `&` does not appear anywhere in the example above. How then did the number read get into the variable `inch`? The answer is that the input operations are defined using reference arguments. A reference is an alternative name for an object. For example, given an integer

```
int i;
```

we can give it a new name `r` like this:

```
int& r = i;
```

The type `int&` is read "reference to `int`" and use of a reference is synonymous to use of the name of the object it was initialized with. For example:

```
i = 7;
r = 7;
```

both assign 7 to `i`. By declaring an argument to be of type reference the classical "call by reference" is obtained.

Class istream is defined like this:

```
class istream {
        // ...
public:
        ostream* tie(ostream& s);

        istream& operator>>(char*);                    // string
        istream& operator>>(char&);                    // single character
        istream& operator>>(short&);
        istream& operator>>(int&);
        istream& operator>>(long&);
        istream& operator>>(float&);
        istream& operator>>(double&);

        istream& get(char* p, int n, int =´\n´&0377);  // string
        istream& get(char& c);                         // single character

        istream& putback(char c);
        // ...
};
```

The operator>> input functions are defined in this style:

```
istream& istream.operator>>(char& c) {
        // skip whitespace
        int a;
        // somehow read a character into a
        c = a;  // this assignment affects the caller
}
```

Naturally constructors and destructors are provided for the type istream as they were for the type ostream.

One problem remains about why the example worked as intended: How did the output stream cout figure out that it needed to write the prompt "inches=" before the input operation took place? After all, had input and output been independent there would have been no reason to flush cout's buffer until the end of the program, and the program did not contain an explicit flush(). The stream I/O library uses the standard technique of "tying" an output stream to an input stream. This means that cin knows about cout and executes a

```
cout.flush()
```

before attempting to read characters from its device. The operation tie() can be used to tie any output stream to any input stream. For example

```
cin.tie(mystream);
```

would cause cin to flush the output stream mystream instead of cout.

Consider the functions

```
istream& operator>>(char*);      // string
istream& operator>>(char&);      // single character
```

The first reads a whitespace terminated string into a vector of characters; the second reads a single character into a char.

The functions reading floating point constants also accept plain integers.

Whitespace and raw input

The ⟩⟩ operator functions all skip whitespace characters. Whitespace is defined as the standard C whitespace by a call to isspace() as defined in ⟨ctype.h⟩. On an implementation using the ASCII character set this defines whitespace to be the characters blank, tab, newline, vertical tab, formfeed, and return.

Where it is not a good idea to simply treat any sequence of whitespace characters as a token separator the functions

```
istream& get(char& c);                          // single character
istream& get(char* p, int n, int ='\n'&0377);  // string
```

can be used. They treat whitespace characters like other characters. The first reads a single character into its argument. the second reads at most n characters into a character vector starting at p. The optional third argument is used to specify a character that will not be read. Default, the second get() function will read at most n characters, but not more than a line: '\n' will not be read.

Stream states

Every stream has a "state" associated with it, and errors and non-standard conditions are handled by setting and testing this state appropriately. The fundamental reason for choosing this approach over the traditional C approach of returning an illegal value in case of trouble was the desire to treat all types (including user-defined types) in the same way, and for many types there is no possible "illegal value" that can be returned. For example, when reading an int, and returning an int every possible return value represents a legal value, so there is no way of representing end-of-file.

An istream can be in one of the following states:

```
enum stream_state { _good, _eof, _fail, _bad };
```

If the state is _good or _eof the previous input operation succeeded. If the state is _good the next input operation might succeed, otherwise it will fail. If one tries to read into a variable v and the operation fails the value of v should be unchanged (it is unchanged if v is of one of the types "known to" class ostream). In other words, applying an input operation to a stream that is not in the _good state is a null operation. The difference between the states _fail and _bad is subtle, and only really interesting to implementors of input operations: In the state _fail it is assumed that the stream is uncorrupted and that no characters have been "lost". In the state _bad all bets are off.

One can examine the state of a stream like this:

```
switch (cin.rdstate()) {
case _good:
        // the last operation on cin succeeded
        break;
case _eof:
        // at end of file
        break;
case _fail:
        // some kind of formatting error
        // probably not too bad
        break;
case _bad:
        // BAD
        break;
}
```

It might be worth noting that if someone invented a new state so that the test above only handled 4 out of 5 cases the compiler would issue a warning.

For any variable z of a type for which the operators >> and << have been defined a copy loop can be written like this:

```
	while (cin>>z) cout << z << "\n";
```

For example, if z is a character vector this loop will take standard input and put it one word (that is, a sequence of non-whitespace characters) per line onto standard output.

When a stream (or a stream operation returning a reference to a stream as in the copy example) is used as a condition, the state of the stream is tested and the test "succeeds", that is the value of the condition is non-zero, (only) if the state is _good. To find out why a loop or test failed one can examine the state.

To copy characters (including whitespace characters) the raw input function get() can be used:

```
	char ch;
	while (cin.get(ch)) cout<<ch;
```

Input of user-defined types

An input operation can be defined for a user-defined type exactly as an output operation was, but for an input operation it is essential that the second argument is of reference type. For example:

```
	istream& operator>>(istream& s, complex& a)
	/*
		input formats for a complex; "f" indicates a float:
			f
			( f )
			( f , f )
	*/
	{
		double	re = 0, im = 0;
		char	c = 0;

		s>>c;
		if (c == '(') {
			s>>re>>c;
			if (c == ',') s>>im>>c;
			if (c != ')') s.clear(_bad);	// set the state
		}
		else {
			s.putback(c);
			s>>re;
		}

		if (s) a = complex(re,im);
		return s;
	}
```

Despite the scarcity of error handling code this will actually handle most kinds of errors well. The local variable c was initialized to avoid having its value accidentally '(' after a failed operation, and the final check of the stream state ensures that the value of the argument a is changed only if everything went well.

More work is needed on the input operations. In particular it would be nice if one could specify input in terms of a pattern (as in languages like Snobol[3] or Icon[4]) and then just test for success and failure of the complete input operation. Such operations would naturally have to provide some extra buffering so that they could "restore an input stream to its original state" after a failed pattern-match operation.

String manipulation

Traditionally the functions sprintf() and sscanf() have been used to do I/O-like operations on character strings. Using streams similar operations can be done by binding an istream or an ostream to a character vector and then using the associated operators exactly as if the stream was bound to a device. For example, if a vector buf contains a traditional zero-terminated string of characters the copy loop presented above can be used to print the words from that vector:

```
char buf[SOMESIZE];
// fill buf
istream ist(sizeof(buf),buf);                // make a stream for buf
char b2[MAX];                                // larger than largest word
while (ist>>b2) cout << b2 << "\n";
```

The terminating zero character is interpreted as end-of-file in this case.

Another use of this would be to read a file into a vector of characters replacing every sequence of whitespace characters with a single space:

```
char buf[SOMESIZE];                          // hopefully large enough
ostream ost(sizeof(buf),buf);
char b2[MAX];                                // larger than largest word
while (cin>>b2) ost << b2 << " ";
```

There is no need to check for overflow of buf; its associated stream ost knows its size and will go into _fail state when it is full.

Another look at the output paradigm

Looking at the examples of output above one might conjecture that "an object should not be printed by some general function, but rather print itself given an output stream and maybe also some formatting information as arguments". In other words the output operator for a type X should look something like this:

```
class X {
        // ...
        print(ostream& s = cout, format_type& format = default_format);
};
X obj;
// ...
obj.print(cerr);
```

This does have some appeal, but could not be the basic model in C++ since built-in types like int are not classes so that it is not possible to write

```
123.print(cerr);          // illegal
```

Furthermore, this style could easily lead to the verbosity that the << operator style of output was invented to avoid:

```
"x = ".print(cerr);       // illegal and verbose
x.print(cerr);
"\n".print(cerr);
```

There are, however, cases where this "inversion" of the output paradigm becomes necessary†.

† The considerations and concepts involved in this example will be familiar to users of Simula67[1], Smalltalk[2], or C++; but may appear rather strange to others. If so, please have a look at any of these languages: the issues are fundamental.

Consider a class shape providing the general concept of a geometric shape:

```
        class shape {
                point center;  // every shape has a center
                // ...
        public:
                // ...
                virtual draw();
        };
```

A shape can be "drawn", that is, printed on a stream, but the function draw() is virtual. That is, a separate draw() function is provided for each particular kind of shape "derived" from class shape. For example:

```
        class circle : public shape {
                int radius;     // a circle has both a center and a radius
        public:
                // ...
                void draw();
                    circle(point cen, int rad);
        };
```

That is, a circle has all the attributes of a shape, and can be manipulated as a shape, but it also have some special properties that must be taken into account when it is manipulated. For example:

```
        shape* p;
        circle c(point(0,0),10);
        p = &c;         // the compiler does not know that *p is a circle
        // ...
        p->draw();      // somewhere else
```

must draw p as a circle. In other words, the fact that the shape pointed to by p is a circle must be deduced at run time from information stored in each shape.

Now consider providing this facility within the *"ostream<<object"* paradigm for output. Like Smalltalk and Simula67, C++ only provides the run time type resolution necessary to determine the actual type of an object at run time using the *"object.operation(argument)"* paradigm, so an << operation must be "inverted". This is how the inversion can be done:

```
        ostream& operator<<(ostream& s, shape* p) {
                return p->draw(s);
        }

        for (shape* p = slist.first(); p; p=slist.next()) cout<<p;
```

Naturally, the reason for the inversion is to maintain a single (terse) paradigm for output operations. Since there is no standard way of passing formatting information on to the virtual output function (draw in this example), the solution is not perfect.

Buffering

The I/O operations have been specified without any mention of device types, but not all devices can be treated identically with respect to buffering strategies. In particular, an ostream bound to a character string needs a different kind of buffer from an ostream bound to a file. There is also a need for double buffering of streams connected to network facilities. These problems are handled by providing different buffer types for different streams at the time of initialization (note the three constructors for class ostream presented above). There is only one set of operations on these buffer types, so the ostream functions do not contain code distinguishing them. However, the functions handling buffer underflow and overflow are virtual. This is sufficient to cope with the buffering strategies needed to date, and an excellent example of the use of virtual functions to allow uniform treatment of logically equivalent facilities with different implementations. The declaration of a stream buffer in ⟨stream.h⟩ looks like this:

```
struct streambuf {                  // a buffer for streams

    char* base;                     // beginning of buffer
    char* pptr;                     // next free byte
    char* gptr;                     // next filled byte
    char* eptr;                     // first byte following buffer
    char  alloc;                    // set if buffer is allocated by "new"

    virtual overflow(int c =EOF); // Empty a buffer.
                                    // Return EOF on error
                                    //        0 on success

    virtual int underflow();        // Fill a buffer
                                    // Return EOF on error or end of input
                                    //        next character on success


    int snextc()                    // get the next character
    {
        return (gptr==pptr) ? underflow() : *++gptr&0377;
    }

    // ...
};
```

Note that the pointers needed to maintain the buffer are specified here so that the common "per character" operations can be defined (once only) as maximally efficient inline functions. Only the overflow() and underflow() functions need to be implemented for each particular buffering strategy. For example:

```
struct filebuf : public streambuf {      // a stream buffer for files

    int fd;                         // file descriptor
    char        opened;             // non-zero if file has been opened

    int overflow(int c =EOF);       // Empty a buffer.
                                    // Return EOF on error, 0 on success

    int underflow();                // Fill a buffer.
                                    // Return EOF on error or end of input
                                    //         next character on success
    // ...
};
```

Efficiency

One might expect that since this I/O facility is defined using generally available language features, it is noticeably less efficient than a built-in facility. This does not appear to be the case. Inline expanded functions are used for the basic operations (like "put a character into a buffer"), so the basic overhead tend to be one function call per simple object (integer, string, etc.) written (or read) plus one function call per buffer overflow. This does not appear to be fundamentally different from other I/O facilities dealing with objects at this level.

Conclusion

It is possible to provide an I/O facility using general language level "data abstraction" facilities in such a way that it compares favorably with a built-in I/O facility in both convenience of use, flexibility, and efficiency. The I/O facility presented here provides a single type secure paradigm for input and output of both built-in and user-defined types. Operator overloading (surprisingly) turned out to be an important tool for avoiding the verbosity traditionally associated with extensible type-secure I/O schemes. The I/O facility handles a range of buffering strategies elegantly, and new strategies can be trivially added. It also needs a bit more work, especially in the areas of formatted output and pattern matching input.

Acknowledgements

Many people have helped me in the design of the stream I/O library, and in the design of the C++ language facilities used in their implementation, notably Brian Kernighan, Doug McIlroy, and Jon Shopiro. A special thanks to Dave Presotto who one day got so disgusted with the efficiency of my original implementation that a new and better one was suddenly in place next morning. That is the implementation described above. In accordance with the best theories of data hiding and data abstraction the user interface had remained constant despite a radical change in the implementation strategy.

References

[1] G.Birtwistle et.al.: SIMULA BEGIN
Studentlitteratur, Lund. 1973.

[2] A.Goldberg and D.Robson: SMALLTALK-80 The language and its implementation
Addison Wesley. 1983.

[3] R.E.Griswold et.al.: The Snobol4 Programming Language.
Prentice-Hall. 1970.

[4] R.E.Griswold and M.T.Griswold: The ICON Programming Language.
Prentice-Hall. 1983.

[5] B.W.Kernighan and D.M.Ritchie: The C Programming Language.
Prentice-Hall. 1978.

[6] Bjarne Stroustrup: Data Abstraction in C.
AT&T BLTJ vol 63, no 8 October 1984 pp 1701-1732.

[7] Bjarne Stroustrup: The C++ Programming Language - Reference Manual.
AT&T Bell Laboratories CSTR-108. January 1984.

[8] Bjarne Stroustrup: The C++ Programming Language.
Addison Wesley. To appear fall 1985.