I/O Stream Manipulator for Binary Integers

Document #:	P00000R0
Date	2025-02-19
C++ Programming Language	
Audience	LEWG

Reply-To Javier Estrada <javier.estrada@outlook.com>

1 Introduction

The **`std::dec`**, **`std::oct`** and **`std::hex`** standard manipulators for integrals numbers address decimal, octal and hexadecimal bases, input/output, respectively. However, there is no current solution for binary input/output, nor for binary literals.

2 Motivation and Scope

The goal and motivation of this paper is to fill a gap in the I/O streams representation without resorting to non-standard solutions. A solution that takes advantage of the existing I/O infrastructure is preferred.

Consider the following motivating example:

```
[Example 1:
#include <bitset>
#include <iostream>

int main() {
  // current solution for output, no solution for input
  std::cout << "The number 42 in binary: " << std::bitset<8>{42} << '\n';
  return 0;
}
— end example]
```

The “bitset trick”, as it is known, only takes care of output, without consideration for width or formatting, showing or hiding the binary base.

A standard solution would make use of a `std::bin` manipulator as follows:

```
[Example 2:
#include <iomanip>
#include <iostream>

using std::setw, std::setfill, std::showbase;

int main() {
  // take advantage of existing infrastructure
  std::cout << setw(8) << setfill('0') << showbase;

  std::cout << "The number 42 in binary: " << std::bin << 42 << '\n';
  return 0;
}
— end example]
```

A similar example for binary input:

```
[Example 3:
#include <iomanip>
#include <iostream>

int main() {
  std::istringstream is("101010");
  is >> std::bin >> n;

  std::cout << "Parsing \"101010\" as binary: " std::bin << n << '\n';
  return 0;
}
— end example]
```

3 Impact on the Standard

This is purely a library addition, requiring no changes to the language

4 Wording Changes

The following sections and tables are affected

4.1 Section 30.4.3.3.3

Before

Table 110: Integer conversion [tab:facet.num .put.int]

State	`stdio` equivalent
`basefield == ios_base::oct`	`%o`
`(basefield == ios_base::hex) && !uppercase`	`%x`
`(basefield == ios_base::hex)`	`%X`
for a `signed` integral type	`%d`
for an `unsigned` integral type	`%u`

After

Table 110: Integer conversion [tab:facet.num.put.int]

State	`stdio` equivalent
`(basefield == ios_base::bin) && !uppercase`	`%b`
`(basefield == ios_base::bin)`	`%B`
`basefield == ios_base::oct`	`%o`
`(basefield == ios_base::hex) && !uppercase`	`%x`
`(basefield == ios_base::hex)`	`%X`
for a `signed` integral type	`%d`

for an **unsigned** integral type	**%u**

4.2 Section 31.5.1 Header <ios> synopsis

A new **basefield** would be added:

```
      // 31.5.5.3, basefield
+     ios_base& bin (ios_base& str);
      ios_base& dec (ios_base& str);
      ios_base& hex (ios_base& str);
      ios_base& oct (ios_base& str);
```

4.3 Section 31.5.2.1 General [ios.base.general]

A new format flag would be added in the class declaration for **fmtflags**:

```
      // 31.5.2.2.2, fmtflags
      using fmtflags = T1 ;
      static constexpr fmtflags boolalpha = unspecified ;
+     static constexpr fmtflags bin = unspecified ;
      static constexpr fmtflags dec = unspecified ;
      static constexpr fmtflags fixed = unspecified ;
      static constexpr fmtflags hex = unspecified ;
```

4.4 Table 118: fmtflags effects [tab:ios.fmtflags]

A new row is added before the **boolalpha** element to account for the **ios_base::bin** flag.

Before

Table 118: Integer conversion [tab:facet.num .put.int]

Element	Effect(s) if set
boolalpha	insert and extract the `bool` type in alphabetic format
Rest of the table stays as is.	

After

Table 118: Integer conversion [tab:facet.num .put.int]

Element	Effect(s) if set
bin	converts integer input or generates integer output in binary base
boolalpha	insert and extract the **bool** type in alphabetic format
Rest of the table stays as is.	

4.5 Section 31.5.5.3 `basefield` manipulators

Two paragraphs are added after paragraph 7 to describe the **`std::bin`** manipulator:

```
ios_base& bin(ios_base& str);
```

8. *Effects*: Calls **`str.setf(ios_base::bin, ios_base::basefield).`**
9. *Returns*: **`str.`**

4.6 Section 31.7.7

Paragraph 4 would add an additional flag to check:

`unspecified` **`setbase(int base);`**

4. *Returns*: An object of unspecified type such that if **`out`** is an object of type **`basic_ostream<charT, traits>`** then the expression **`out << setbase(base)`** behaves as if it called **`f(out, base)`**, or if **`in`** is an object of type **`basic_istream<charT, traits>`** then the expression **`in >> setbase(base)`** behaves as if it called **`f(in, base),`** where the function **f** is defined as:

```
   void f(ios_base& str, int base) {
     // set basefield
-    str.setf(base == 8 ? ios_base::oct :
+    str.setf(base == 2 ? ios_base::bin :
+    base == 8 ? ios_base::oct :
     base == 10 ? ios_base::dec :
     base == 16 ? ios_base::hex :
     ios_base::fmtflags(0), ios_base::basefield);
   }
```

The expression **`out << setbase(base)`** has type **`basic_ostream<charT, traits>&`** and value **`out`**.
The expression **`in >> setbase(base)`** has type **`basic_istream<charT, traits>&`** and value **`in`**.