BitStream API

Ali Khudiyev
NOT COMPLETED

Contents

1 Introduction

BitStream is a single header C++ library that has been developed to beautify programmer's experience when dealing with lots of bit manipulations. My current reason to develop such library is to be able to avoid bit manipulation parts of the compression algorithms as much as possible, and if you are someone who is intensively implementing such algorithms then you might end up losing a lot of time due to code repetition which usually brings up bug repetition as well. This is the reason that I have tried to make this library easy and flexible enough to work with it without noticing significant counter-intuitiveness against to C/C++ programming languages.

This documentation is to help you with the library functionalities, and the bit_stream.h has also nice comments to guide you through it.

Here is a small example to start with...

```
#include "bit_stream.h"
#include <cstdio>

int main(){
    BitStream bs;
    bs.push<char>('A', bs.begin());
    char c = bs.get<char>(bs.begin());
    return 0;
}
```

Best Practice!
You do not need to necessarily obey what is written in here, however, it is recommended to do so in order to avoid bugs and misinterpretations.

Be careful!
You need to pay attention to and obey what is mentioned in this section, otherwise, you will end up with bugs and crashes.

Remember!
Kind reminders for you to avoid potentially misleading or confusing concepts.

Important!
Important notes that are essential for you to understand to build intuition or grasp general ideas behind the described content.

2 Functional Specification

There a bunch of things that the programmer has to know before using the BitStream library. The library has a way of managing memory and there is rules that are guaranteed to be respected any time a library function is invoked. Once the programmer understands the simple mechanics of the memory management and setter/getter implementation logic then all the library is going to become easy and flexible hopefully.

2.1 Memory management

For better performance, the library uses a vector of 8-bit unsigned integers (uint8_t from *stddef.h*) each of which holds 8 bits given by the user optimally. To be more specific, if n bits have been given then only $\lceil \frac{n}{8} \rceil$ 8-bit unsigned integers. It may actually be more than that due to STL implmentation of vectors, however, it is not the concern for now and in subsequent sections, a bunch of methods are going to be introduced to deal with such cases easily. Now, let's look at the memory management in the library.

Vector indices		0(vector.begin)	1	2	3	(vector.end)
Memory address		0	+1B	+2B	+3B	
Forward flow		+3i	+2i	+1i	0	
Reverse flow		0	+1i	+2i	+3i	
Stream indices	stream.end	3	2	1	0(stream.begin)	

Table 1: Memory management

The table 1 is to illustrate that if the flow direction of bit stream is set to be *forward* then the bits are stacked from higher(right) memory address to lower(left) memory address and vice versa. So, if the user decides to send bits **011** (first bit '0', then double '1's), the bit '0' will occupy the highest memory address while the last bit '1' will occupy the lowest memory address. Also note that, this library has been developed in a *little endian* computer.

2.2 Read & Write operations

Whenever a user wants to push some bits to the stream, (s)he needs to provide an iterator besides the bits. The concept of iterators are not new in C++ at all, and I have developed as much similar as possible concepts for the Bit Stream library. There are two main types of iterators when it comes to the bit flow direction: **forward** and **reverse** iterators. When **forward iterators** are used then the given bits flow from higher to lower memory addresses, and it is the opposite in case of **reverse iterators**.

Let's look at an 8-bit unsigned integer, an empty stream and how the integer is pushed into the stream.

8-bit uint	b7	b6	b5	b4	b3	b2	b1	b0	
stream.begin	b7	b6	b5	b4	b3	b2	b1	b0	
stream.rbegin	b0	b1	b2	b3	b4	b5	b6	b7	
	stream.begin								**stream.end**
stream.rend								**stream.rbegin**	

Table 2: Setting bits in Bit Stream from 8-bit uint

All the bits are fetched from the 8-bit source from the most significant to the least significant. As individual bits are fetched, two things happen consecutively - current bit is put into the stream by the provided iterator, and the iterator is incremented.

	stream.begin								**stream.end**
stream.rend								**stream.rbegin**	
	b7	b6	b5	b4	b3	b2	b1	b0	
stream.begin	b7	b6	b5	b4	b3	b2	b1	b0	
stream.rbegin	b0	b1	b2	b3	b4	b5	b6	b7	

Table 3: Getting bits from Bit Stream as 8-bit uint

As in STL containers, the following (in)equalities hold for the Bit Stream as well:

$$\text{stream.begin} < \text{stream.end}$$
$$\text{stream.rbegin} < \text{stream.rend}$$
$$\text{stream.begin} = \text{stream.rend-1}$$
$$\text{stream.rbegin} = \text{stream.end-1}$$

Remember!
SET writes bits in memory beginning from the lowest address and the most significant bit (and ending with the highest address and the least significant bit) to a bit stream beginning from a given iterator.
GET writes bits in a bit stream beginning from a given iterator into memory beginning from the lowest address and the most significant bit (and ending with the highest address and the least significant bit).

Important!
Bits are read/written from/to memory beginning from the most significant to the least significant bit of the bytes beginning from the lowest to the highest address.

The figure shown below illustrates 2-byte integer (short) and the corresponding byte and bit addresses. The bytes stored in a little endian computer are reversed when fetched from memory. For example, if there is a pointer ptr pointing to 2-byte unsigned integer, then assigning ptr [0] = 0xb0 and ptr [1] = 0x2e would result with an integer value of 0x2eb0 instead of 0xb02e.

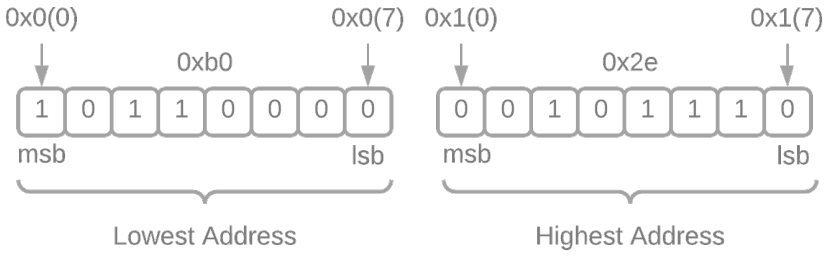

Figure 1: Bit Stream Processing in Memory

When the **SET** action is performed with this integer and the begin() [1]/rbegin() iterator [2], the bit stream will look like this:

[1]		1	0	1	1	0	0	0	0	0	0	1	0	1	1	1	0	
		0x00							0x07	0x10							0x17	
	rend	**begin**															**rbegin**	**end**
		0x17							0x10	0x07							0x00	
[2]		0	1	1	1	0	1	0	0	0	0	0	0	1	1	0	1	

Table 4: Example bit stream

Be Careful!
As obvious, the bytes in little endian architectures are stored in a reversed order. So, when you try to **GET** 2 bytes from the previous streams [1] and [2], you are going to get the integer values of 0x2eb0 and 0xb02e respectively.

3 API Specification

The interface to interact with the *bit streams* is quite simple and intuitive. There is no library-defined "namespace"s or any complex structure that the programmer should be aware of. There are 5 main types of utilities in the library and each of them has been introduced to support the design paradigms of C++ standard libraries:

1. Constructor & Destructor
2. Iterators
3. Setters & Getters
4. Modifiers
5. Operators

3.1 Constructor & Destructor

The *BitStream* class has several constructors that are shown below:

```
BitStream();
BitStream(const uint8_t* const source,
          size_t count,
          size_t offset,
          bool forward);
BitStream(size_t n, bool bit);
BitStream(const std::string& bit_chars);
```

BitStream(const uint8_t* const, size_t , size_t , bool) is used to initialize a bit stream from some memory source. It writes **count** [number of] bits starting from the **offset** bit to the bit stream in the given [**forward=true**/ **false**] direction.

BitStream(size_t, bool is used to initialize the bit stream with **n** [number of] same bits that are equal to **bit**.

BitStream(const std::string& is used to initialize the bit stream according to the characters of the given string **bit_chars** whose values have to be either 1 or 0, otherwise, the value is skipped.

3.2 Iterators

There are several functionalities of iterators that can be a "program-saver" in hard situations. There are 2 main properties of iterators: *constantness* and *reverseness*. **Constant iterators** are used to read bits from the stream whereas **regular** or **non-constant iterators** are used to read/write bits from/to the stream. In addition to this, **reverseness of iterators** is used to determine which way operations are carried on. Since the *BitStream* library only supports 1D stream of bits, any iterator can be either reverse or forward. Functional specification of iterators are described in the subsection 3.2.4. Here are the member functions of *BitStream* class to obtain iterators:

```
BitStream::iterator begin();
BitStream::const_iterator cbegin();
BitStream::reverse_iterator rbegin();
BitStream::const_reverse_iterator crbegin();

BitStream::iterator end();
BitStream::const_iterator cend();
BitStream::reverse_iterator rend();
BitStream::const_reverse_iterator crend();
```

Iterators are used mostly as the way they are used in STL libraries - *reading*, *writing*, *erasing* or for any other modifications which are going to be explained in the section 3.4. They are really important to understand since almost every bit stream functionality in the API works with them. Iterators have also their own standalone functionalities grouped by:

1. Setter & Getter
2. Modifiers

3. Operators

This functionalities can be used independently as soon as you get an iterator from a bit stream. That is to say, every iterator that you request for carries information of the bit stream to which it is referring when performing operations. There are also several situations that previously requested iterators get invalidated because of some critical changes made upon the relevant bit stream. In such cases, it is the programmer's job to keep track of iterators and operations carried on.

3.2.1 Setters & Getters

Iterators, depending on their constantness, can be used to access and/or modify individual bits in the relevant stream. If the iterator is constant then it can only be used to read individual bits and if non-constant, it can also be used to write bits besides reading them. Here are the most basic operations that could be performed by iterators:

```
void set(bool bit);
bool get() const;
```

Be careful!
In order to set a bit, you have to use a non-const iterator.

3.2.2 Modifiers

There are 4 main functions to obtain the current state of an iterator:

1. Accessibility - whether iterator element is accessible
2. Constantness - whether iterator is const_iterator or const_reverse_iterator
3. Reverseness - whether iterator is reverse_iterator or const_reverse_iterator
4. Validity (special cases)

```
bool is_accessible() const;
bool is_const() const;
bool is_reverse() const;
bool is_valid() const;
```

3.2.3 Operators

There are a bunch of overloaded operators which can make the use of iterators even more relevant and easy. All the usable operators are just addition and subtraction of one form or the other and an iterator does not need to be non-const object in order to be used any postfix/prefix operators or even operators such as += or −= since a programmer has to be able to iterate through his bit stream even if it is restricted to read-only access. Here are the definitions for operators:

```
const xxx_iterator operator+(size_t offset) const;
const xxx_iterator operator-(size_t offset) const;

xxx_iterator operator+(size_t offset);
xxx_iterator operator-(size_t offset);

xxx_iterator& operator+=(size_t offset);
xxx_iterator& operator-=(size_t offset);

size_t operator-(const xxx_iterator& it) const;

// Prefix operators
iterator& operator++() const;
iterator& operator--() const;

iterator& operator++();
iterator& operator--();
```

```
// Postfix operators
iterator operator++(int);
iterator operator--(int);
```

3.2.4 Functional specification

The memory management of a bit stream is shown below:

Memory address	+0	+1B	+2B	+3B	+4B
Index	0	1	2	3	4
Forward stream	→	d	c	b	a
Reverse stream	a	b	c	d	←
Forward iterator offset	4	3	2	1	0
Reverse iterator offset	0	1	2	3	4

Table 5: Bit stream in memory

Relative memory address	-2B	-1B	0	+1B	+2B
Regular iterator	*end()*	*rbegin()*		*begin()*	*rend()*
Constant iterator	*cend()*	*crbegin()*		*cbegin()*	*crend()*
Stream	.	uint8_t	uint8_t	uint8_t	.

Table 6: Bit stream iterators

The addition and subtraction operators are overloaded for the iterators in an intuitive way such that a programmer need not care about the low-level memory addresses at all. The programmer should be easily able to traverse the stream forwards and backwards just by using the iterator abstraction. The abstraction is that any *end iterator* minus that of *begin iterator* is always greater than 0. So, if a programmer were to traverse the stream from low to high memory address, the loop of for(auto it=rbegin(); it!=rend(); ++it) could be used and if the desired behaviour is the opposite, the loop of for(auto it=begin(); it!=end(); ++it) could be used instead.

Best Practice!
WHENEVER YOU ARE USING **reverse** AND **forward** ITERATORS SIMULTANEOUSLY AND DEPENDENTLY TO MANIPULATE A BIT STREAM, IT MEANS THAT THERE IS PROBABLY A BETTER OR LESS ERROR-PRONE OR LESS CONFUSING WAY OF DOING THE SAME THING. IT IS BETTER TO RECONSIDER THE WHOLE CASE OF DEPENDENT USE OF THESE 2 DIFFERENT TYPES OF ITERATORS AGAIN!

3.3 Setters & Getters

Beside iterators, the **BitStream** class has also *setters* and *getters*. These access modifiers are used in a more generalized manner for manipulation of chunk of bits all together. To simulate these behaviours that the generalized setters and getters perform, one would manually set or get individual bits with iterators which can cause a lot of pain since you usually want to either read specific chunks from memory and store them in a regular (non-BitStream) object or write such (non-BitStream) chunks into a bit stream. In other words, you sometimes want to cast a bit stream to a non-stream custom class object and vice-versa, and that is why generalized setters and getters are important. Here are the list of BitStream in-class setters and getters:

```
template<typename T>
size_t set(const T& bits,
           constant_iterator_type it,
           size_t count,
           size_t offset);

template<typename T, size_t count=1>
size_t set(const T* const src,
           constant_iterator_type it,
           size_t count,
           size_t offset);
```

```
bool set(bool bit,
         const_iterator_type it);
```

Be Careful!
Iterators that are to be provided for setters have to be either of type **iterator** or **reverse_iterator** .

Remember!
Templated setters are usually used to cast a literal or custom object to a bit stream.

```
template<typename T>
T get(iterator_type it,
      size_t count,
      size_t offset) const;

template<typename T>
size_t get(T* const dest,
           iterator_type it,
           size_t count,
           size_t offset) const;

bool get(iterator_type it) const;
```

Remember!
Templated getters are usually used to cast a bit stream to a literal or custom object.

3.3.1 Functional specification

Memory address	-1B	0	+1B	+2B	+3B
src	.	a	b	c	.
Forward iterator	.	c	b	a	.
Reverse iterator	.	a	b	c	.

Table 7: Writing bits from memory chunk to a bit stream

3.4 Modifiers

3.4.1 assign

Assigning is used to reinitialize the bit stream with a given source of bit(s). In fact, what it does is to call **reset-set** functions. Since user has to provide the number of bits that are going to be assigned, *reset* function also allocates that number of bits for the *set* function to be able to just copy data from the source without managing memory dynamically and inefficiently. Here are the function declarations:

```
template<typename T, bool forward>
void assign(const T& bits,
            size_t count,
            size_t offset);

template<typename T>
void rassign(const T* const src,
             size_t count,
             size_t offset);
```

3.4.2 push / insert / pop

```
template<typename T>
void push(const T& bits,
          dynamic_iterator_type it,
          size_t count,
          size_t offset);
```

```
template<typename T>
void push(const T* const src,
          dynamic_iterator_type it,
          size_t count,
          size_t offset);

void push(bool bit,
          dynamic_iterator_type it);
```

```
template<typename T>
void insert(const T& bits,
            dynamic_iterator_type it,
            size_t count,
            size_t offset);

template<typename T>
void insert(const T* const src,
            dynamic_iterator_type it,
            size_t count,
            size_t offset);

void insert(bool bit,
            dynamic_iterator_type it);
```

```
void pop(dynamic_iterator_type it,
         size_t offset);

void pop(dynamic_iterator_type start,
         dynamic_iterator_type stop);
```

Be Careful!
The pop function that expects two dynamic iterators as arguments, it has to be provided with only either forward or reverse iterators. That is to say, you cannot pass both forward and reverse iterators simultaneously.

3.4.3 rotate / shift / flip

```
void rotate(size_t n,
            dynamic_iterator_type start,
            dynamic_iterator_type stop);

void rotate(size_t n,
            dynamic_iterator_type it,
            size_t offset);

void rotater(size_t n);

void rotatel(size_t n);
```

```
void shift(size_t n,
           dynamic_iterator_type start,
           dynamic_iterator_type stop);

void shift(size_t n,
           dynamic_iterator_type it,
           size_t offset);

void shiftr(size_t n);

void shiftl(size_t n);
```

```
void flip(dynamic_iterator_type start,
          dynamic_iterator_type stop);

void flip(dynamic_iterator_type it,
          size_t offset);
```

Be Careful!
For rotators and shifters that expect two dynamic iterators as arguments, they should be provided with only either forward or reverse iterators. That is to say, you cannot pass both forward and reverse iterators simultaneously.

3.4.4 memcpy / memmov / memflp

memcpy copies **n** bits from **dest** to **src**. If the memory sections intersect with each other then the behaviour is undefined.

memmov moves **n** bits from **dest** to **src**. Unlike *memcpy*, if the memory sections intersect, the behaviour is not undefined and the bits are moved or "copied" without any problem.

memflp flips **n** bits starting from **it**.

Be Careful!
memcpy is fast but vulnerable to intersecting memory sections whereas **memmov** is slow but cautious to such intersections.

```
void memcpy(size_t n,
            dynamic_iterator_type src,
            constant_iterator_type dest);

void memmov(size_t n
            dynamic_iterator_type src,
            constant_iterator_type dest);

void memflp(size_t n,
            dynamic_iterator_type it);
```

Remember!
These in-stream memory operations do not require iterators to be the same type. That is to say, you can provide **reverse_iterator** and **const_iterator** simultaneously as long as the former is of type dynamic and the latter is of type constant.

3.4.5 substream / cast

```
BitStream substream(iterator_type start,
                    iterator_type stop);

BitStream substream(iterator_type it,
                    size_t offset);
```

```
template<typename T>
T cast_to(size_t count,
          size_t offset);

template<typename T>
BitStream& cast_from(const T& bits,
                     size_t count,
                     size_t offset);

template<typename T>
BitStream& cast_from(const T* const src,
                     size_t count,
                     size_t offset);

template<typename T> static
```

```
BitStream& cast_from(const T& bits,
                     size_t count,
                     size_t offset);

template<typename T> static
BitStream& cast_from(const T* const src,
                     size_t count,
                     size_t offset);
```

3.5 Operators

```
bit_proxy operator[](size_t offset);
const bit_proxy operator[](size_t offset) const;

BitStream operator~() const;
BitStream operator&(const BitStream&) const;
BitStream operator|(const BitStream&) const;
BitStream operator^(const BitStream&) const;
BitStream operator<<(size_t n) const;
BitStream operator>>(size_t n) const;

BitStream& operator&=(const BitStream&);
BitStream& operator|=(const BitStream&);
BitStream& operator^=(const BitStream&);
BitStream& operator<<=(size_t n);
BitStream& operator>>=(size_t n);

bool operator==(const BitStream&) const;
bool operator!=(const BitStream&) const;
bool operator<(const BitStream&) const;
bool operator>(const BitStream&) const;
bool operator<=(const BitStream&) const;
bool operator>=(const BitStream&) const;

BitStream operator+(const BitStream&) const;
BitStream operator-(const BitStream&) const;

BitStream& operator+=(const BitStream&);
BitStream& operator-=(const BitStream&);
```

3.6 Functional specification

Obviously, there is no bit-level access in the most of the architectures such as x86-64, MIPS and even LC-3. So, it is harder to manipulate individual bits as a programmer since the previously mentioned architectures do not have single bit fetch support and even if some did, it does not get exposed to a programmer on the most of ISAs either. However, most ISAs allow programmers to use binary operators such as <<(left shit), >>(right shift), !(not), ^(xor), &(and), |(or) and so on. Only by using these operators, a programmer can manipulate the individual bits without directly fetching them one-by-one. The **BitStream** library also uses these operators heavily to for overall performance boost (memory + speed). **BitStream** class definition is shown below:

```
class BitStream{
    private:
    size_t m_bit_count;
    std::vector<uint8_t> m_bytes;
    ...
};
```

3.6.1 Memory

The bit stream object stores the bits as a chunks of bytes. The STL library vector container is used for this exact purpose.

3.6.2 Speed

...

4 Usage and examples

4.1 How to use the BitStream library

1. Download the header file bit_stream.h
2. Include the header file in your project file(s)

4.2 Examples

```
#include "bit_stream.h"
#include <cstdio>

struct BitChunk{
    char c[4];
};

int main(){
    BitChunk bc { 0x41, 0x42, 0x43, 0x44 };

    BitStream bs;
    bs.push<BitChunk>(bc, bs.begin());
    auto chunk = bs.get<BitChunk>(bs.begin());

    printf("%c %c %c %c\n",
        chunk.c[0], chunk.c[1], chunk.c[2], chunk.c[3]);

    return 0;
}
```