P6-BitArray

CS 3370 – C++ Software Development

Program 6

“Dynamic Bit Strings”

This assignment uses bitwise operations and operator overloading. You will let the **vector** class template do all of the memory management for you, however (whew!). Users look upon instances of this class as bit strings, indexed left-to-right like any other type of string.   Do not think of these as large numbers.

Create a class named **BitArray** that holds a dynamic number of bits, packed into unsigned integers to save space.   You will use a vector of integers to store the bits. It is up to you to process bits in the correct position inside the appropriate integer in the vector.

You need to keep track of how many integers are needed to hold the bits in use. As bits are appended, you will have to expand to the vector if it is full (i.e., if all bits are currently used; vector::resize and/or vector::push_back are handy for this). In addition to providing the capability to set, reset, and test individual bits, this class provides some convenient operations commonly needed in bit-related applications.

Remember, the bits appear to the user as if they are bit *strings*, not numbers, so if the BitArray **b** holds 1011, then the 0th, 2nd, and 3rd bits are 1 and **b[1]** is 0.

We will make the underlying integer type of the array a template parameter, which defaults to **size_t**, so your class definition will look like this:

```
template<class IType = size_t>
class BitArray { … };
```

On most platforms, this means that you can hold 32 or 64 bits in each array element (I’ll call the individual integers “blocks”), but don’t hard code 32 or 64 throughout your code. Instead, you can calculate the number of bits per block at compile time inside your class as follows:

```
enum {BITS_PER_WORD = CHAR_BIT * sizeof(IType)};
```

The macro CHAR_BIT is defined in **<climits>**, and is the number 8 on most platforms (duh).

The most efficient way to store bits in each integer may surprise you. To better understand the layout of a **BitArray**, suppose a **BitArray** object currently tracks 84 bits and **sizeof(size_t)** == 4 bytes. Then BITS_PER_WORD would be 32 and the array would need = 3 integer elements to hold 84 bits, and the “last” 12 bits in the third block would be unused. For simplicity in accessing bits, we will have bit 0 of the **BitArray** be bit 0 in the 0th integer block in the array, as the following diagram of bit positions illustrates:

31 30... ... 1 0	63 62... ... 33 32	(unused bits) 83 82... ... 65 64

Although this layout does not reflect the logical order users visualize for a bit string, it makes easy work for you of setting and resetting bits by bitwise operations. Suppose, for example, a user has a **BitArray** object, **b**, and wants to *set* (i.e., turn “on”) the bit in position 50:

```
b[50] = 1;          // Uses BitArray::operator[](size_t)
```

To implement this, your **operator[ ]** needs to determine that this bit position resides in array element 1 (the second “block”), and is bit number 18 offset from the right in that block. This, of course, is very easy:

```
block = bitpos / BITS_PER_WORD;   // 1 = 50/32
offset = bitpos % BITS_PER_WORD; // 18 = 50%32
```

You can then define the appropriate mask and change the second block (**words[1]**) in your array.

Naturally, users expect the bits to be processed *as if* they were stored positionally in increasing index order, *left-to-right* (so bit-0 is logically *left-most*), so any I/O functions (like **operator<<** and **operator >>**) should process them in that string-like order.

See the driver ***main.cpp* (https://uvu.instructure.com/courses/489107/files/94257649/download)** for examples.

Class Definition

The class interface you need to implement follows.

```
template<class IType = size_t>

class BitArray {

public:

    // Object Management
    explicit BitArray(size_t = 0);
    explicit BitArray(const string&);
    BitArray(const BitArray& b) = default;                  // Copy constructor
    BitArray(BitArray&& b) noexcept;                        // Move constructor

    BitArray& operator=(const BitArray& b) = default;       // Copy assignment
    BitArray& operator=(BitArray&& b) noexcept;  // Move assignment

    size_t capacity() const;                          // # of bits the current allocation can hold

    // Mutators

    BitArray& operator+=(bool);                       // Append a bit
    BitArray& operator+=(const BitArray& b);          // Append a BitArray

    void erase(size_t pos, size_t nbits = 1);         // Remove “nbits” bits at a position

    void insert(size_t, bool);                        // Insert a bit at a position (slide "right")
    void insert(size_t pos, const BitArray&);         // Insert an entire BitArray object

    void shrink_to_fit();                            // Discard unused, trailing vector cells

    // Bitwise ops

    bitproxy operator[](size_t);                      // bracket operator, sets or gets bit via BitProxy
    bool operator[](size_t) const;                    // Const version, required for bracket operator.  No need for BitProxy

    void toggle(size_t);                              // Toggles a single bit
    void toggle();                                    // Toggles all bits

    BitArray operator~() const;                       // ~ operator toggles/reverses all bits, returning a temp

    BitArray operator<<(unsigned int) const;         // Shift operators…
    BitArray operator>>(unsigned int) const;
    BitArray& operator<<=(unsigned int);
    BitArray& operator>>=(unsigned int);

    // Extraction ops
    BitArray slice(size_t pos, size_t count) const;    // Extracts a new sub-array

    // Comparison ops
    bool operator==(const BitArray&) const;
    bool operator!=(const BitArray&) const;
    bool operator<(const BitArray&) const;
    bool operator<=(const BitArray&) const;
    bool operator>(const BitArray&) const;
    bool operator>=(const BitArray&) const;

    // Counting ops

    size_t size() const;                             // Number of bits in use in the vector
    size_t count() const;                            // The number of 1-bits present
    bool any() const;                                // Optimized version of count() > 0

    // Stream I/O (define these in situ)
    friend ostream& operator<<(ostream&, const BitArray&);
    friend istream& operator>>(istream&, BitArray&);

    // String conversion
    string to_string() const;
};
```

Remember that all of your code should reside in a header file (*bitarray.h*; that's how templates work). If you refer to the **BitArray** *type* outside of the class definition, or if you create a local **BitArray** object, you must qualify it as **BitArray**<**IType**>.

Note that all single-argument constructors (except the copy constructor) are **explicit**, so all operator functions can be members (except the stream operators, of course). Define the bodies of the stream operators as friends inside the class definition itself (this is important for templates!).

The first constructor initializes a **BitArray** object to the appropriate number of 0-bits if its argument is greater than zero. The second constructor expects a string consisting only of characters '0' and '1' and builds the corresponding **BitArray** object with the bits set in the same logical configuration.

Stream Operators

Remember that if a stream contains **0100abc**, then the stream input operator (>>) will overwrite its **BitArray** argument with **0100** and leave **abs** in the stream, just like reading numbers does. Throw a **runtime_error** (declared in **<stdexcept>)** if any other characters occur in the input argument string (even whitespace). An empty string is okay, though (just create an empty object). For the input operator, set the stream to a fail state if there are no valid bit characters to be read in the input stream (after skipping white space, of course).

Bracket Operators/BitProxy

Throw a **logic_error** (also declared in **<stdexcept>**) if any out-of-range indexing is attempted anywhere.

Define a nested, private **bitproxy** class to accommodate intelligent use of **operator[ ]**, as discussed in class (i.e., distinguish between **b[i] = true** and **bool val = b[i]**). See ***bits.cpp*** **(https://uvu.instructure.com/courses/489107/files/96193754/download)** in the code set for hints on how to define it.

Other Program Notes

Move Construction/Assignment - You have two options with regards to how you leave the state of the 'moved' object:

1. Leave the 'temp' (moved) object in an initial state, eg., size() ==0, cleared vector etc.
2. Swap the internal contents completely between the two objects.

The **shrink_to_fit** function eliminates any unused blocks at the end of your storage vector that may have accrued after calls to **erase** (so you are, in effect, supporting an on-demand, “shrink-to-fit” storage allocation policy, but not forcing it). Just call vector::resize appropriately.

The comparison operators should compare **BitArray** objects *lexicographically* (i.e., as if they were strings, in dictionary order). The rest of the member functions should be self-explanatory.

Don't convert your BitArray to a string using to_string() to implement comparisons or other operators such as insert, erase etc. It is possible, but defeats much of the purpose of the exercise.

Professional tip: Begin by implementing some private, low-level functions to handle individual bits in any position. See Program 6 section of **Program Hints and Caveats page (https://uvu.instructure.com/courses/489107/pages/program-hints-and-caveatshttps://uvu.instructure.com/courses/489107/pages/program-hints-and-caveats)** for a list of these functions. They can then be used to great advantage in writing other functions.

Testing and Submission

There is a file, *test.h*, and a test program, main.cpp that you can download from Zylabs (or **files->Program 6 (https://uvu.instructure.com/courses/489107/files/folder/Programs/Program%206#)**). Use these to test your code before you turn submit to Zylabs. The zylabs tests are based on the unit tests in main.cpp. It is far easier to run and test locally before submitting to Zylabs. The main.cpp output should look like:

```
move constructor

move assignment

move assignment

move assignment

move assignment

move assignment

move assignment

shrinking from 2 to 1 words

Test Report:

    Number of Passes = 69

    Number of Failures = 0
```

Note the trace statements above the report. You should output 'move assignment', 'move constructor' in the move functions. shrink_to_fit should output as above also.

FYI, my *bitarray.h* is about 350 lines of executable code.

Grading/Rubric

- See our general **Grading Criteria for CS 3370 Programs (https://uvu.instructure.com/courses/489107/pages/grading-criteria-for-cs-3370-programs)** for general deductions
- Program 6 will be submitted in Zylabs
  - Points awarded(100 pts total)
    - 18 Unit Tests totaling 100 points
- Deductions
  - Should not have explicit dependencies on type of integer used for the backing vector (e.g., 1u) -3
  - Ineffective us of std::vector methods to manage the size of the underlying vector of integers (especially resize) -2
  - Effective implementation of overloaded operators
    - did not write some operators in terms of other operators; -2
      - << should be written in terms of <<=, not visa versa -2
    - did not have low-level, private member functions for locating, reading, and writing individual bits); -3
    - A lot of extra move assignments from <<=, >>=, insert ops. should not use move assignment, just act on bitarray itself, e.g., resize, then read/assign. -4
    - excessive use of strings and temp objects for internal operations -3
- Extra Credit
  - Extra Credit is applied from second examplary requirement onwards.
    - Example, zylabs score 92, 1st exemplary req (+0), 2nd exemplary req (+2) = 94
    - Max score is capped at 100%
  - Exemplary Items
    - Simplest possible logic; utility functions like read_bit and assign_bit (and others) are used. +2
    - Correct implementation of move semantics;+2
      - Move Constructor leave internal contents of moved items in safe state (either swapped with left hand side, or zeroed out)
    - Efficient implementation of bitproxy class and the associated BitArray index operators +2

Assessment Rubric

Competency ↓	**Emerging →**	**Proficient →**	**Exemplary**
Memory Management		Effective use of **std::vector** methods to manage the size of the underlying vector of integers (especially **resize**)	
Clean Code	No magic numbers (use named constants and inline functions for program parameters)	No repeated code (refactor); No unnecessary code	Simplest possible logic to fulfill program requirements; utility functions like **read_bit** and **assign_bit** (and others) are used.
Defensive Programming	Exceptions thrown as requested		Use **assert** in appropriate places (I only have one, actually, in **read_bit**)
Templates	Use a single type parameter for the underlying integer type	No explicit dependencies on type of integer used for the backing vector	

Other		Effective implementation of overloaded operators (beware repeated code; write some operators in terms of other operators; have low-level, private member functions for locating, reading, and writing individual bits); set stream state appropriately in operator>>	Correct implementation of move semantics; efficient implementation of bitproxy class and the associated BitArray index operators