Lab 01 - C++

Course CSI2372, Fall 2016
University of Ottawa

CSI 2372 Lab Assignment 1

In this lab assignment, you will write a program that represents binary numbers as an array of an enumerated type. You also will need convert to and from the corresponding bitsets.

Create the source file verbose_binary.cpp and the header file verbose_binary.h

Create a scoped enumeration VerboseBinary in the header file with the constants corresponding to the power of two: One,Two,Four,Eight,Sixteen as well as Null. (You will need to decide on the appropriate type to use in the enumeration).

In *.h

```
enum  !?!  VerboseBinary  :  !?!  ;
```

In *.cpp:

```
enum  !?!  VerboseBinary  :  !?!  {
One,Two,Four,Eight,Sixteen as well as Null  !?!
};
```

Book, p.1018, 19.3. Enumerations

Enumerations let us group together sets of integral constants. Like classes, each enumeration defines a new type. Enumerations are literal types (§ 7.5.6, p. 299).

C++ has two kinds of enumerations: scoped and unscoped. The new standard introduced **scoped enumerations**. We define a scoped enumeration using the keywords **`enum class`** (or, equivalently, **`enum struct`**), followed by the enumeration name and a comma-separated list of **enumerators** enclosed in curly braces. A semicolon follows the close curly:

```
enum class open_modes {input, output, append};
```

We define an **unscoped enumeration** by omitting the class (or struct) keyword. The enumeration name is optional in an unscoped enum:

```
enum color {red, yellow, green}; // unscoped enumeration
// unnamed, unscoped enum
enum {floatPrec = 6, doublePrec = 10, double_doublePrec = 10};
```

By default, enumerator values start at 0 and each enumerator has a value 1 greater than the preceding one. However, we can also supply initializers for one or more enumerators:

Click here to view code image

```
enum class intTypes {
charTyp = 8, shortTyp = 16, intTyp = 16,
longTyp = 32, long_longTyp = 64
};
```

As we see with the enumerators for **intTyp** and **shortTyp**, an enumerator value need not be unique. When we omit an initializer, the enumerator has a value 1 greater than the preceding enumerator.

```
#ifndef VERBOSE_BINARY_H_
#define VERBOSE_BINARY_H_

#include <bitset>

using std::bitset;

// functions declarations here

#endif
```

p.944, **Defined Terms**

bitset Standard library class that holds a collection of bits of a size that is known at compile time, and provides operations to test and set the bits in the collection.

p.211, 4.8. The Bitwise Operators

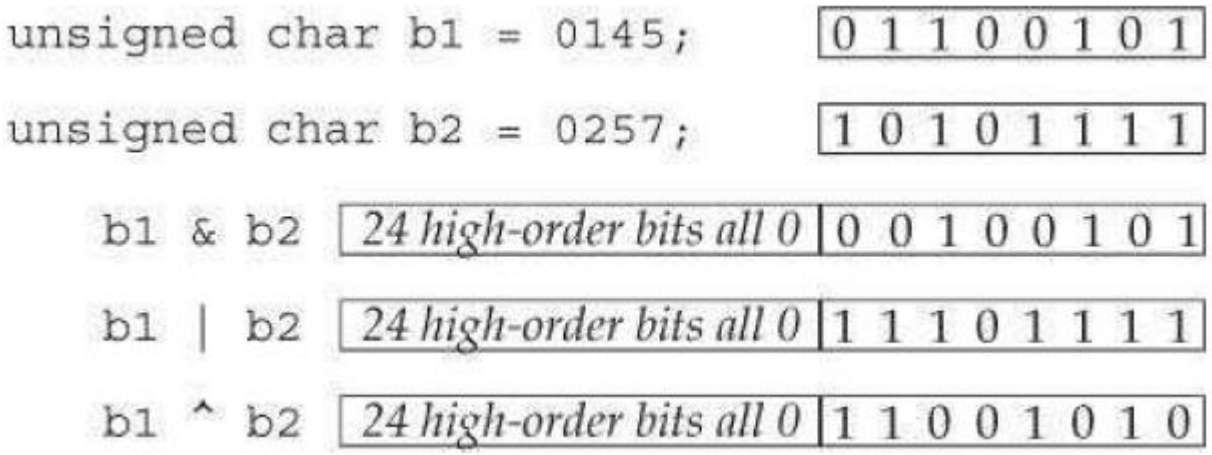
The bitwise operators take operands of integral type that they use as a collection of bits. These operators let us test and set individual bits. As we'll see in § 17.2 (p. 723), we can also use these operators on a library type named bitset that represents a flexibly sized collection of bits.

As usual, if an operand is a "small integer," its value is first promoted (§ 4.11.1, p. 160) to a larger integral type. The operand(s) can be either signed or unsigned.

Operator	Function	Use
~	bitwise NOT	`~expr`
<<	left shift	`expr1 << expr2`
>>	right shift	`expr1 >> expr2`
&	bitwise AND	`expr1 & expr2`
^	bitwise XOR	`expr1 ^ expr2`
|	bitwise OR	`expr1 | expr2`

p.213, Bitwise AND, OR, and XOR Operators

The AND (&), OR (|), and XOR (^) operators generate new values with the bit pattern composed from its two operands:



p.890, Table 17.2. Ways to Initialize a bitset

`bitset<n> b;`	`b` has `n` bits; each bit is 0. This constructor is a `constexpr` (§ 7.5.6, p. 299).
`bitset<n> b(u);`	`b` is a copy of the `n` low-order bits of `unsigned long long` value `u`. If `n` is greater than the size of an `unsigned long long`, the high-order bits beyond those in the `unsigned long long` are set to zero. This constructor is a `constexpr` (§ 7.5.6, p. 299).
`bitset<n> b(s, pos, m, zero, one);`	`b` is a copy of the `m` characters from the `string` `s` starting at position `pos`. `s` may contain only the characters `zero` and `one`; if `s` contains any other character, throws `invalid_argument`. The characters are stored in `b` as `zeor` and `one`, respectively. `pos` defaults to 0, `m` defaults to `string::npos`, `zero` defaults to `'0'`, and `one` defaults to `'1'`.
`bitset<n> b(cp, pos, m, zero, one);`	Same as the previous constructor, but copies from the character array to which `cp` points. If `m` is not supplied, then `cp` must point to a C-style string. If `m` is supplied, there must be at least `m` characters that are `zero` or `one` starting at `cp`.

The constructors that take a `string` or character pointer are `explicit` (§ 7.5.4, p. 296). The ability to specify alternate characters for 0 and 1 was added in the new standard.

p.893, Table 17.3. bitset Operations

`b.any()`	Is any bit in `b` on?
`b.all()`	Are all the bits in `b` on?
`b.none()`	Are no bits in `b` on?
`b.count()`	Number of bits in `b` that are on.
`b.size()`	A `constexpr` function (§ 2.4.4, p. 65) that returns the number of bits in `b`.
`b.test(pos)`	Returns `true` if bit at position `pos` is on, `false` otherwise.
`b.set(pos, v)`	Sets the bit at position `pos` to the `bool` value `v`. `v` defaults to `true`. If no
`b.set()`	arguments, turns on all the bits in `b`.
`b.reset(pos)`	Turns off the bit at position `pos` or turns off all the bits in `b`.
`b.reset()`	
`b.flip(pos)`	Changes the state of the bit at position `pos` or of every bit in `b`.
`b.flip()`	
`b[pos]`	Gives access to the bit in `b` at position `pos`; if `b` is `const`, then `b[pos]` returns a `bool` value `true` if the bit is on, `false` otherwise.
`b.to_ulong()`	Returns an `unsigned long` or an `unsigned long long` with the same
`b.to_ullong()`	bits as in `b`. Throws `overflow_error` if the bit pattern in `b` won't fit in the indicated result type.
`b.to_string(zero, one)`	Returns a `string` representing the bit pattern in `b`. `zero` and `one`

Create a function Bitset<5> convert(!?!) which accepts an array of size 6 of the type of your enumeration and returns the corresponding Bitset<5> value. Unused values in the array should be indicated by a terminating Null (similar idea to old-style c-strings, please see example below). You will have to replace !?! with the correct type. Implement this function in verbose_binary.cpp and declare it in the header verbose_binary.h.

```
bitset<5> convert( !?! ) {
  !?! accu = 0;
  for (int i=0; i<6; ++i ) {
    if ( !?! ) !?!;
    accu += static_cast<int>(vb[i]);  }
  return accu;
}
```

Create a function void convert(!?!, !?!) which accepts an array of size 6 of the type of your enumeration and the corresponding Bitset<5> value. Unused values in the array should be indicated by a terminating Null (similar idea to old-style c-strings, please see example below). You will have to replace !?! with the correct type. Implement this function in verbose_binary.cpp and declare it in the header verbose_binary.h.

2^0=1; 2^1=2; 2^2=4; 2^3=8; 2^4=16; total= 5 bits; add 0;

```
//In *.h file:
void convert(bitset<5> bs, VerboseBinary vb[6]);
//In *.cpp file:
void convert(bitset<5> bs, VerboseBinary vb[6] ) {
  int pos = 0;
  if (bs.test(4)) vb[pos++]=VerboseBinary::Sixteen;
  if
  if
  if
  if
  vb[pos]=VerboseBinary::Null;
}
```

!?!

Unused values in the array should be indicated by a terminating Null (similar idea to old-style c-strings, please see example below).

Example Run:

Enter two numbers between 0-31:
31 17
Sixteen, Eight, Four, Two, One, Null
Sixteen, One, Null
Eight, Four, Two, Null

```
// main.cpp
#include <iostream>
#include <bitset>
#include "verbose_binary.h"
int main() {
  int a,b;
  bitset<5> aBs,bBs;
  std::cout << "Enter two numbers between 0-
31:" << std::endl;
  std::cin >> a >> b;
  if (a<0 || a>31) return -1;
  if (b<0 || b>31) return -2;
  aBs = static_cast<bitset<5>>(a);
  bBs = static_cast<bitset<5>>(b);
  // std::cout << aBs << " " << bBs <<
std::endl;
```

```
// main.cpp
  VerboseBinary aVB[6];
  VerboseBinary bVB[6];
  convert(aBs,aVB);
  convert(bBs,bVB);
  print(aVB);
  print(bVB);
  xorEquals(aVB,bVB);
  print(aVB);
  return 0;
}
/** Do not alter
* Your solution must work with this main program
**/
```

Create a function void print(!?!) that accepts a **VerboseBinary array of size 6** and prints it to console as a binary number. Again, implement this function in verbose_binary.cpp and declare it in the header verbose_binary.h.

```
// in *.h file
void print( !?! );
// in *.cpp file
void print( !?! ) {
  for ( !?! ) {
    if (vb[i]==VerboseBinary::Sixteen) cout <<"Sixteen";
    if
    if
    if
    if
    if (vb[i]==VerboseBinary::Null) {
       !?! }
    cout << ", ";
  }
```

!?!

Create a function void xorEquals(!?!, !?!) which takes two arrays of size 6 of type VerboseBinary. The function should performs a binary xor operation on the two parameters and put the result in the first parameter.

```
// In *.h file
void xorEquals(                !?!                ,
               !?!                );
// in *.cpp file
void xorEquals(                !?!                ,
               !?!                ) {
  bitset<5> res = convert(aVB) ^ convert(bVB);
  convert(res,aVB);
}
```

Test your implementation with the supplied main function in the separate file main.cpp The main function will ask for console input of two integer numbers between 0-31, will print the numbers VerboseBinary to console, then will call the above function xorEquals and print the result to console.

Example Run:

Enter two numbers between 0-31:
31 17
Sixteen, Eight, Four, Two, One, Null
Sixteen, One, Null
Eight, Four, Two, Null

You must hand-in exactly the three files: verbose_binary.cpp and verbose_binary.h in a zip archive. Do not hand-in any other files or use any other type of archive. No word files or pdfs (or similar word processing files) will be accepted.