Exercise 6 Data Types

1. (Value Categories)
Determine by inspection if the following expressions are *xvalue*, *lvalue* or *prvalue*:

a) a ? b : c (ternary conditional expression for some a, b and c.)
b) a+b, a%b, &a.
c) "Hello world".
d) nullptr.
e) ++a, --a.
f) a++, a--.

Can you use type traits to answer this question as well?

2. (Advantages of std::enable_if)
Which of the following statements can be considered useful features of std::enable_if?:

a) More user-friendly error messages than when using 'raw' (unrestricted) template parameters.
b) Its use can lead to more robust code.
c) It restricts templates to types that have certain properties.
d) Its use reduces the amount of boilerplate code that needs to be written.

3. (std::vector<bool> versus std::bitset<>)
An alternative to bitsets is to employ the class std::vector<bool>. There has been much discussion about the shortcomings of this class (for example, it does not necessarily store its elements as a contiguous array).

Answer the following questions:
a) Determine which functionality it supports compared to the two bitset classes discussed here.
b) Create a function to compute the intersection of two instances std::vector<bool>.
Having completed the exercise will probably convince you that it is better to use bitset classes instead of std::vector<bool>?

4. (Creating Object Adapters for Bitset, Compile-Time (Composition))
In this exercise we create a compile-time *bit matrix* (call it BitMatrix<N, M>) consisting of N rows and M columns all of whose elements are bits. Some requirements are:

- The chosen data structure must be efficient (for example, accessing the elements).
- Its interface must have the same look and feel as that of std::bitset<>.
- We wish to reuse as much code as possible.
- It must be generic enough to support a range of applications in different domains (for example, Computer Graphics and its many applications).

Answer the following questions:

a) Determine which data structure to use in order to implement BitMatrix<N, M>, for example as a nested array std::array<std::bitset<M>, N> or a one-dimensional array std::bitset<N*M>. Which choice is "optimal" is for you to decide. You need to determine which criteria to use for example, performance and maintainability.
b) Constructors need to be created. Use the same defaults as with std::bitset<M>.
c) Implement the following operators for all rows in the matrix and for a given row in the matrix:
    - Set/reset all bits.

- Flip the bits.
- Test if none, all or any bits are set.
- Access the elements.
- Count the number of set bits.

d) Create member functions for OR, XOR and AND Boolean operations on bit matrices.
e) Consider create the matrix as a derived class of bitset.

5. (Comparing Singly and Doubly Linked Lists)
In this exercise we carry out some operations on `std::list<double>` (call it A for convenience) and `std::forward_list<double>` (call it B).

Answer the following questions:

a) Create instances of A and B with n elements, where n is typically a large number (for example, at least a million).
b) Insert an element at every alternate position in the lists A and B.
c) Remove all even elements from the lists A and B.
d) Sort and reverse the lists A and B.
e) Create an instance of B with n elements all of whose values are the same value `val`. Compare the run-time efficiency of using a single call to remove all the elements with value `val` and removing elements one-by-one.

Use the stopwatch class to measure the relative run-time performance in all cases.