C++操作符重载教学题：编写bitvector

目标

基于我们给的代码框架，编写一个容器MyBitVector，用该容器维护一个0/1序列

通过这个练习学习：如何使用操作符重载进行设计以简化代码

框架代码概要和要求

- 需要设计两个私有成员变量
  - size 表示存储和打印的0/1序列长度
  - data 表示存储的0/1序列
- 构造函数
  - `MyBitVector(int size)` 存储一个长度为size，内容为0的序列
  - `MyBitVector(const string &s)` 参数为0/1字符串序列，将其转为对应的0/1序列进行存储
- 成员函数
  - `set(int index)` 将index位置的bit设置为1
  - `clear(int index)` 将index位置的bit设置为0
  - `get(int index)` 获取index位置的bit信息
  - `print` 打印一行长度为size的0/1序列，中间没有空格
- 操作符重载
  - 二元操作符：
    - `MyBitVector operator&(MyBitVector &other)` 按位与操作
    - `MyBitVector operator|(MyBitVector &other)` 按位或操作
    - `MyBitVector operator^(MyBitVector &other)` 按位异或操作
  - 单元操作符
    - `MyBitVector operator~()` 按位取反
    - `MyBitVector &operator&=(MyBitVector &other)` 原地按位与操作
    - `MyBitVector &operator|=(MyBitVector &other)` 原地按位或操作
    - `MyBitVector &operator^=(MyBitVector &other)` 原地按位异或操作

测试代码

- 包括main函数和对应的test*()测试用例
- 测试内容为`print`的打印结果，即当前MyBitVector实例中存储的0/1序列

注意事项

- **完成TODO标注的函数**
  - 注意操作符重载的函数签名

        - 有的是返回新的MyBitVector
        - 有的是返回已有MyBitVector的引用
    - 建议在实现get/set/clear时加上运行时检查，防止产生非法越界操作

- 提交要求
    - **请不要修改main函数和测试代码！**可能会影响后台用例的判定
    - 不要投机取巧！
        - 助教会人工检查运行行为异常的代码提交，并将本次练习记录为0分

- 测试样例

```
void test(){
  MyBitVector bv1("00000000");
  MyBitVector bv2("11110011");
  MyBitVector bv3 = bv1 | bv2;
  bv3.print();
}
//正确输出结果
11110011
```

练习之外（不作为练习，仅供扩展学习）

- bitvector有很多良好的特性，在实践中的应用非常广泛
    - 数据存储得非常紧密，能充分利用缓存，减少耗时的内存store/load操作
    - 有利于实现bit-level的并行计算
    - 可以用在需要大量求解交并集的场景中
- 但是bitvector很容易造成大量的空间浪费
    - 一般会根据具体场景设计或使用特定的bitvector实现，可以搜索`sparse bit vector`
- 一个设计良好的bitvector需要考虑很多因素
    - 如何实现高效的扩容
    - 如何对长度不同的bitvector进行位操作
    - 如何减少空间浪费

附录：代码框架

```
#include <iostream>
using namespace std;

class MyBitVector {
private:
    // TODO: Add your private member variables here
```

```
public:
    explicit MyBitVector(int size) {
        // TODO: Add your constructor here, and initialize your
bitvector with 0s
    }

    explicit MyBitVector(const string &s) {
        // TODO: Add your constructor here, and initialize your
bitvector with the given string of 0s and 1s
    }

    void set(int index) {
        // TODO: Set the bit at index to 1
    }

    void clear(int index) {
        // TODO: Set the bit at index to 0
    }

    bool get(int index) {
        // TODO: Return the value of the bit at index
        return false;
    }

    void print() {
        // TODO: Print the bitvector to continuous 0s and 1s of given
length
    }

    // TODO: operator overloads to generate new bitvectors from
existing ones
    MyBitVector operator&(MyBitVector &other) {
        // TODO: bv = bv1 & bv2
    }

    MyBitVector operator|(MyBitVector &other) {
        // TODO: bv = bv1 | bv2
    }

    MyBitVector operator^(MyBitVector &other) {
        // TODO: bv = bv1 ^ bv2
    }

    MyBitVector operator~() {
        // TODO: bv1 = ~bv2
        // question: what if we want to modify the original bitvector
without creating a new one?
    }
```

```
    // TODO: operator overloads to modify existing bitvectors in place
    MyBitVector &operator&=(MyBitVector &other) {
        // TODO: bv1 &= bv2
    }

    MyBitVector &operator|=(MyBitVector &other) {
        // TODO: bv1 |= bv2
    }

    MyBitVector &operator^=(MyBitVector &other) {
        // TODO: bv1 ^= bv2
    }

};

void test1() {
    MyBitVector bv1("01010100");
    MyBitVector bv2("11101111");
    MyBitVector bv3 = bv1 & bv2;
    bv1 &= bv2;
    bv1.print();
    bv2.print();
    bv3.print();
}

void test2() {
    MyBitVector bv1("00001000");
    MyBitVector bv2("11010011");
    MyBitVector bv3 = bv1 | bv2;
    bv1 |= bv2;
    bv1.print();
    bv2.print();
    bv3.print();
}

void test3() {
    MyBitVector bv1("00010010");
    MyBitVector bv2("10111001");
    MyBitVector bv3 = bv1 ^ bv2;
    bv1 ^= bv2;
    bv1.print();
    bv2.print();
    bv3.print();
}

void test4() {
    MyBitVector bv1("00100100");
    MyBitVector bv2 = ~bv1;
    bv1 = ~bv1;
```

```
    bv1.print();
    bv2.print();
}

void test5() {
    MyBitVector bv1("00000001");
    bv1.set(2);
    bv1.print();
}

#define TEST(x) std::cout << "test" << #x << "\n"; test##x();

int main() {
    TEST(1);
    TEST(2);
    TEST(3);
    TEST(4);
    TEST(5);
    return 0;
}
```