实验课 1 比特流基础

Project 1 bitset

背景

计算机中的所有数据都是由 `01` 串的形式存储的。无论是整数、浮点数还是字符串，本质上都是对一段 `01` 串定义不同的操作。我们也可以仿照这个结构，实现一个 `01` 串的维护结构，并在此基础上继承得到整数类和其他类。

实验描述

在实验 1 中，我们需要实现一个能使用的 `bits` 类。在本周课程中我们学习了类函数、类的构造和析构函数，这些都是可以应用到 `bits` 类中的。

为了保证你的所有命名和全局的命名没有冲突，你应当把所有实现的代码放在一个命名空间 `mybits` 中。

为了存储比特，使用 `bool` 类型是非常低效的。因此你需要尝试把比特放在其他的类型中，例如 `unsigned int` 或 `unsigned long long` 。推荐使用 `size_t` 作为存储的载体。

为了方便你对载体的修改，建议使用 using ：

```
using storage_type = size_t;
using storage_pointer = storage_type *;
```

你需要实现的类的类名为 `bits`

- 你应当实现一个默认构造函数，支持创建一个空的 `bits`
- 应当实现通过一个 ULL 创建一个 `bits` 。较低位应当存放在比特流的较低下标。
- `bits( const char* s , size_t n , char zero , char one )` 通过一个字符串创建一个 `bits` 。其中的所有 `zero` 字符会被视作 0 ，所有 `one` 字符会被视作 1 。出现非法字符应当抛出 `invalid_arguments` （或者默认替换为 0）。字符串下标较小的部分应该放在 `bits` 的**高位**。
- 如果你觉得有必要，需要实现一个复制构造函数。
- `bool test( size_t p )` 返回 p 位置的比特。
- `size_t count()` 返回所有比特中 1 的个数。
- `size_t size()` 返回总共占用的内存大小（Byte 数）。

- `bool all() const` 返回是否所有位置均为 1。
- `bool any() const` 返回是否存在位置为 1。
- `bool none() const` 返回是否全为 0 。
- `bits& flip()` 将所有比特翻转，返回当前 bits 的引用。
- `bits& set()` 将所有比特设为 1 ，返回当前 bits 的引用。
- `bits& reset()` 将所有比特设为 0 ，返回当前 bits 的引用。

思考：这里后面几个对 bits 的操作为什么需要返回当前 bits 的引用。

实验指南

原理

由于一个 `bool` 的存储空间为一个字节（byte），而一个字节中有 8 个字（bit），一个字就已经有 0 / 1 状态，这样每一个 `bool` 就浪费了 7 个 bit，这一点在很长的比特流的情况下会有很大的空间浪费。

考虑到一个 `unsigned long long` 为 64 位无符号整数，即在二进制下，一个 `unsigned long long` 为一个有 64 个 bit 的数，用它的每个 bit 来代表我们所写的一个 `bits` 类的一个 bit。我们可以用这样一个数组 / `vector` / ... 等方法存储这样若干个 `unsigned long long`，我们的所有操作都需要对 **二进制** 下的数进行操作，因此需要学习位运算的操作。

为了我们的类能够具有更广泛的适用性，我们并没有使用 `unsigned long long`，而是使用 `size_t` 这样的类型。同时由于这样的类型在不同的设备上位数不同，我们可以用 `sizeof(size_t) * 8` 得到 `size_t` 的位数。

注意：这里我们不要把一个数看成一个十进制数，把它看成一个二进制数会更有助于理解。

步骤

如果你觉得实现一个这样的类不知道从何下手，那么可以把问题分解一下：

1. 定义一个数组或者利用 STL 的容器，存储所有比特的值。
2. 定义三个构造函数，它们需要满足三种分别的功能。
3. 实现给出的基本方法。

分别考虑三个部分的实现方案。

存储

首先，每个整数在计算机中都是通过二进制存储的。也就是说每个整数本身就是一段零一串。想要存储任意长度的零一串，其实就是存储任意长度的整数。

有三种基础的存储的方案可供选择。如果你有更好的方案也可以使用。

静态数组

```
const int N = 1000;
class bits {
private:
  storage_type data[N];
  size_t size;
};
```

直接使用静态数组对所有的比特存储。这种方法在学习模版类后可以变得更好。

指针与动态分配空间

```
class bits {
private:
  storage_type *data;
  size_t size;
public:
  bits( int n ) : size( n ) { data = new storage_type[n]; }
};
```

使用 STL 容器存储

```
class bits {
private:
  std::vector<storage_type> data;
};
```

在学习模版类后，你可以尝试让你的类变成一个 STL 类的 wrapper 。

构造函数

你需要保证构造函数被调用后所有的类中的数据都有了确定的初始值。

函数实现

在函数实现前，你需要学习一般整数的**位运算**。你可以询问 google 或者 ChatGPT 。

例如，如果你需要对所有比特做翻转，那么其实就是对所有存储的整数执行零一翻转，也就是

```
bits& flip( ) {
    size_t n = N;
    storage_pointer p = data;
    for( ; n >= bits_per_word ; ++p , n -= bits_per_word )
        *p = ~*p;
    if( n > 0 ) {
        storage_type m = ( ~storage_type( 0 ) >> ( bits_per
_word - n ) );
        storage_type b = ( *p ) & m;
        *p = ( ~b ) & m;
        *p = ( ~*p ) & m;
    }
    return *this;
}
```

例如，翻转某一位时，需要寻找到这一位并执行单位的翻转，找到这个位所属的那一个数，然后对这个数的该位进行翻转。

本次实验中的几乎所有方法都需要枚举每一个整数并尝试通过位运算和 `storage_type` 的基本操作得到结果。