第三次上机

题目描述

如果你了解TCP，你可能知道可靠字节流的抽象在通信中是很用的，即使Internet本身只提供“尽力而为”（不可靠）数据报的服务，但如果你从未听说过TCP也没关系，本次上机将会实现一个简易的ByteStream：(你可以默认输入的是char类型)，你会实现一个提供这种抽象的对象：字节写在“输入”端，可以从“输出”端以相同的顺序读取。字节流是有限的：**写端通过`end_input()`可以结束输入，然后再也不能写入字节,如果写端结束输入且读端读取到流的末尾时（两个条件需要同时满足，请注意），字节流将达到“EOF”，并且不能读取更多的字节。**字节流对象用一个特定的“容量”(`capacity`)初始化，**字节流将限制写入程序在任何给定时刻的写入量，以确保流不会超过其存储容量。当读端读取字节并将其从字节流pop时，允许写端写入更多字节。**你实现的字节流只会用于单个线程，所以不必担心并发。

需要说明的是：字节流是有限的，但在写端结束输入并完成流之前，它几乎可以任意长（在到达`EOF`之前可以进行多次`write`）。你的实现必须能够处理比容量长得多的流。容量由`capacity`限制，但不限制流的长度。容量只有一个字节的对象仍然可以实现一个GB甚至1个TB长的流（比如写端写一个`char`，读端读取一个`char`，写端再写，读端再读）。

写端需要解决的接口如下：

```
// Write a string of bytes into the stream. Write as many
// as will fit, and return the number of bytes written.
size_t write(const std::string &data);
// Returns the number of additional bytes that the stream has space for
size_t remaining_capacity() const;
// Signal that the byte stream has reached its ending
void end_input();
// Indicate that the stream suffered an error
void set_error();
```

读端的接口如下所示：

```
// Peek at next "len" bytes of the stream,just peek not pop!!!!(so we use
const for this function)
std::string peek_output(const size_t len) const;
// Remove ``len'' bytes from the buffer
void pop_output(const size_t len);
// Read (i.e., copy and then pop) the next "len" bytes of the stream
```

```
std::string read(const size_t len);
bool input_ended() const;// `true` if the stream input has ended
bool eof() const;// `true` if the output has reached the ending
bool error() const;// `true` if the stream has suffered an error
size_t buffer_size() const;// the maximum amount that can currently be
peeked/read
bool buffer_empty() const;// `true` if the buffer is empty
size_t bytes_written() const;// Total number of bytes written
size_t bytes_read() const;// Total number of bytes popped
```

合起来：

```
class ByteStream {
private:

    //一端读 一端写，先进先出
    //容器：
    //这里我们推荐您使用deque双端队列作为容器，尽管可能ByteStream行为上
    //更类似于queue队列(他们都是先进先出)，但是可惜的是为了实现可靠性(读出一些数据却不
会丢失数据，也就是peek_output接口)，
    //我们需要能够对容器进行迭代(queue并不具有迭代器)。
    //如果您不了解deque的用法，您也可以使用char数组和两个指针来表示队列的头和尾
    //一些必要的变量：用来表示内存容量，已经写入的字节数，已经读出的字节数，输入流是否结
束，输出流是否结束
    //size_t用来表示无符号整型，您的整型变量的类型均可设为size_t
    bool _error{};  //表明stream是否出现了error

    //需要您格外注意：输入流和输出流结束的判断，输入流结束的时候会调用end_input()通
知，
    //在输入流结束且容器中全部都读出之后，整个流到达eof
public:
    //构造函数
    ByteStream(const size_t capacity);

    //将string类型的data写入stream中，返回写入的字节数
    size_t write(const std::string& data);
    //返回remaining_capicity
    size_t remaining_capacity() const;
    //输入流发出信号 表明到达输入流结尾
    void end_input();
    //! stream出错
    void set_error() { _error = true; }
    //获取stream输出流的一部分
    std::string peek_output(const size_t len) const;
    //对输出流pop
    void pop_output(const size_t len);
    //read len的数据，可以理解为两步peek+pop
    std::string read(const size_t len);
    //返回bool变量
    bool input_ended() const;
    //stream出错
    bool error() const { return _error; }
```

```
    //返回当前容器的size
    size_t buffer_size() const;
    //如果容器为空返回true
    bool buffer_empty() const;
    //如果输入流到达EOF，返回true
    bool eof() const;
    //返回总共写入的字节数
    size_t bytes_written() const;
    //返回总共读出的字节数
    size_t bytes_read() const;
};
#endif  // SPONGE_LIBSPONGE_BYTE_STREAM_H
```

Tips

- 建议大家看完函数接口的描述之后，先去看下测试样例文件`test.cc`。
- 如果你的结果和`assert`中描述的不一致，则会触发异常，此时你可以选择断点调试或者在代码中插入`print`语句进行调试。
- 虽然函数的参数是`len`，但由于`capacity`的限制，不一定能写入/读出`len`。
- `end_input()`和`input_ended()`一个发出信号表明输入流结束，一个返回`bool`类型变量表示输入流是否结束。
- `eof()`表示输出流是否结束：意味着输入流结束并且容器中没有多余的字节可读
- 如果要使用`deque<char>`，要么使用`std::deque<char>`，要么在`.hh`文件中加上`#include<iostream>`，`using namespace std;`
- `set_error()` 和 `bool error() const` 是给上一层对象调用的接口，所以在该实验中并不需要考虑，只需要给定初始值然后返回就好。
- `write()`也是给上一层对象调用的接口，根据返回值来实现可靠传输的，想想看这是为什么？

提交要求

- 提交源码文件：`byte_stream.cc` 和 `byte_stream.hh` ,直接打包成zip格式的压缩包。**不要添加其他任何目录，压缩包中只包含这两个个文件!**
- `.hh`，`.cc`和`.h`，`.cpp`一样，都是告诉编译器该文件是由`c++`写成的，为了在`OJ`上编译不出错，请保证您的后缀名为`.hh`，`.cc`。
- 文件的编码格式只支持utf-8。
- 请严格按照给定的接口进行编码,否则无法调用测试用例。
- **提交的源码文件中不需要包含main函数,否则无法通过编译。**

附录

begin()	返回指向容器中第一个元素的迭代器。
end()	返回指向容器最后一个元素所在位置后一个位置的迭代器，通常和 begin() 结合使用。
rbegin()	返回指向最后一个元素的迭代器。
rend()	返回指向第一个元素所在位置前一个位置的迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
size()	返回实际元素个数。
max_size()	返回容器所能容纳元素个数的最大值。这通常是一个很大的值，一般是 2^{32}-1，我们很少会用到这个函数。
resize()	改变实际元素的个数。
empty()	判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。
shrink _to_fit()	将内存减少到等于当前元素实际所使用的大小。
at()	使用经过边界检查的索引访问元素。
front()	返回第一个元素的引用。
back()	返回最后一个元素的引用。
assign()	用新元素替换原有内容。
push_back()	在序列的尾部添加一个元素。
push_front()	在序列的头部添加一个元素。
pop_back()	移除容器尾部的元素。
pop_front()	移除容器头部的元素。

- 迭代器的使用：

```
for (auto it = vec_.begin(); it != vec_.end(); it++) {
    //这里用*it来获得对应位置的元素
}
```

- 如果你需要用`deque<char>q`的前n个char构成一个`string`，你也可以这样写 `std::string(q.begin(),q.begin()+n)`