CS 3370 – C++: Programming Assignment 1

Prof. Todd W. Flyr (and other faculty)
14th January, 2026

A General Requirements for All Programming Assignments

- All code must be submitted separately in Canvas along with a link to your project in OnlineGDB. (Later I may add Github Codespaces, but not for this assignment.)
- All code must build in OnlineGDB or your project will returned ungraded and will receive late days.
- Any code that is deemed extraneous or excessive may lower your grade.
- If you are unable to complete any portion of the assignment before it is due, you must make a note of it when you turn it in or it may not be graded. If there is a note, discuss what works, what doesn't, and what you think is wrong. This way the grader can be more lenient.
- Reflection: summarize what you learned and problems that you overcame in finishing this project. Put this in a comment block in your source code or in a separate file in the project or text in the Canvas submission. (5 points out of 100). Note that every assignment requires this and it is important as it gives the instructor important feedback about what you learn and what assignment may be improved.
- Per the syllabus: 10% off per class day (M-F) late. You have two grace days you use at any time but you must request them when the assignment is turned in. They may not be used retroactively.
- You may consult AI **only** after you've completed all aspects of the code yourself but are maybe stuck on something. You may **not** at any time copy code from AI into your project.
- Any assignments that the grader deems too substantially similar to be coincidence will either receive a zero or a grade divided between all of the similar assignments, regardless of the reason.
- See Rubric Section below for specific grading details.

Advice: Every assignment in this course from here on out will almost certainly take multiple days to complete. Do not wait until a day or two before it is due to work on it.

B Program 1 – A Dynamic Circular Buffer

B-1 Overview

Because output devices (screen, printer, disk, etc.) are generally slower than the processors, data to be output is often buffered – put into memory where the output device can get it whenever it is ready. Because these buffers are used over and over, they are sometimes implemented as a circular buffer.
In a circular buffer, you (the processor) start at the front, and add data. In the meantime, the output device would be pulling data off the buffer behind you. When you reach the end of the buffer, you go back to the beginning and continue to fill. Of course, this assumes that the output device has already cleared out the data at the beginning of the buffer! (More on that later.)

B-2 Implementation Requirements

You implement a Circular Buffer class. Inside the class, you will have a dynamic (heap) array. Keep integer indexes (or pointers, if you prefer), for the head and the tail of the data in use. As you insert characters into the buffer, you will advance the head index, and as you remove characters, you advance the tail index. When either reaches the end of the buffer, it will wrap around to the beginning.

B-2-1 Buffer Growth

What happens when you want to add more characters than the buffer has room? You have to grow the internal array. Allocate a new array which is one `CHUNK` larger than the previous one. (`CHUNK` is currently defined as 8). Copy the data from the old array into the new one, preserving the order of the data, of course. While the data could start somewhere in the middle of the old array, and even wrap around the end, just start the data at the beginning of the new array. Be sure to clean up; don't leak memory!
Note that you have a `size()`, which is the number of elements currently in the buffer. You also have `capacity()`, which is the maximum number of elements you can currently have without growing the buffer. For example, if your buffer has only `'X'` in it, the size is 1, but the capacity is 8.

B-2-2 Constructor

The constructor takes an argument which is the number of elements you want to reserve space for. So if the argument is 5, the capacity will be 8; however, the initial size will still be 0. Note that if the argument is 0, the initial capacity will be 0; you will not allocate any space initially.

B-2-3 Elements and Operations

The elements of the Circular buffer are characters. There are insert functions for a single character, and for multiple characters, with a number of how many characters to read. For convenience, there is a string version as well. For removing characters, there is a `get` function that gets a single character, and one that gets multiple characters. For convenience, it returns them as a string. If you request more characters than are in the buffer, you get all of them.
For the purposes of verifying the correctness of your implementation, there is a function called `examine()`. It returns a string representing the contents of the buffer, enclosed in the square brace characters. Represent unused elements with `'-'`.
Note: when you remove an element from the buffer, **IT IS INCORRECT TO REPLACE THAT CHARACTER WITH A** `'-'`. DO NOT change the element in the buffer; just advance the tail index/pointer.

B-2-4 Buffer Shrinking

It may be nice to occasionally shrink the buffer. Implement a function that reduces the size of the buffer. It's kind of the reverse of growing the buffer.

B-3 Class Specification

```
#include <string>
using std::string;

class CircBuf {
    const size_t CHUNK { 8 };
    // Insert your data and private functions here.

public:
    // Number of elements you want it to be able to hold to
    ↪ start with.
    CircBuf(size_t reserve = 0);
    ~CircBuf();
    size_t size();
    size_t capacity();
    void insert(char);
    void insert(const char*, size_t sz);
```

```
    void insert(const string&);
    char get();
    string get(size_t);
    // Returns a string with all the characters, AND shrinks the
     ↪  buffer to zero.
    string flush();
    string examine();
    void shrink(); // Reduces the unused space in the buffer.
};
```

B-4 Submission Requirements

Implement your `CircBuf` class in a file called `CircBuf.cpp`. `CircBuf.h` will need to be modified as it is only an interface with no member data.

This project has a unit test harness `test.h` that you see is included into the `main.cpp` file and includes a large number of tests. Put all of these into OnlinGDB.cpp and build them. **You may not modify *test.h* or *main.cpp*.**

You must pass all the test cases in `main.cpp` to get full credit. Note, however, that the grader reserves the right to run additional test cases against your program.

Professional Tip: When you craft a program, you not only must solve the problem, but you must also write code that presents the solution clearly to a reader of your code. Before you turn your program in, make an edit pass through it with the idea that you may need to explain it to someone else (or to yourself in the future :-).

C Assessment Rubric

General Grading Applying to All Assignments

- **A Grade**: meets all execution requirements implemented in a manner meeting all coding requirements in the assignment and rubric below with at most a minor mistake or two.
- **B Grade**: meets execution requirements but has at most one significant requirement missing either in the code or in the input/output behavior.
- **C Grade**: missing more than one significant execution requirement and coding requirement, or multiple minor requirements not met along with one significant execution requirement.
- **D Grade or lower**: missing more than two major requirements in any category there are substantial with the codebase, but the project still builds.
- Any project that does not build in OnlineGDB will be replied to via the grader that it does not build and that late days count when fixing it. It will remain ungraded until it builds.

Competency	Basic	Proficient	Exemplary	Points
Memory Management	Every `new` has a corresponding `delete` (no memory leaks); destructor does the right thing; Use `delete[]` when deleting heap arrays.	Data is properly recentered when growing the array	Memory efficiency	10
Memory Efficiency		Every free space in the array is available to the user (follow the spec for `front_` and `back_`)		5
Clean Code	No magic numbers (use named constant for `CHUNK`); use simple, meaningful variable names	No repeated code (refactor); No unnecessary code	Simplest possible logic to fulfill program requirements; Use `std::copy` instead of a loop when growing the array (Don't use `memcpy`!)	5
Other	Use `size_t` for non-negative quantities	Always `#include <cstddef>` for `size_t`; prefer `#pragma once` to include guards `#ifndef`–`#endif`	Use `std::` (either as a prefix or in a using declaration) in your header file for names imported from the standard library (never use `using namespace std` in header files)	5

Table 1: Assessment Rubric