CST 8152 – Compilers - Assignment #1

Due Date: prior or on October 3td, 2018

Earnings: 6% of your course grade (plus 1% bonus)

Purpose: Programming and Using Dynamic Structures (buffers) with C

This is a review of and an exercise in C coding style, programming techniques, data types and structures, memory management, and simple file input/output. It will give you a better understanding of the type of internal data structures used by a simple compiler you will be building this semester. This assignment will be also an exercise in "*excessively defensive programming*". You are to write functions that should be "overly" protected and should not abruptly terminate or "**crash**" at run-time due to invalid function parameters, erroneous internal calculations, or memory violations. To complete the assignment, you should fulfill the following two tasks:

Task 1: The Buffer Data Structure and Utility Functions

Buffers are often used when developing compilers because of their efficiency (see page 111 of your textbook). You are to implement a buffer that can operate in three different modes: a "**f**ixed-size" buffer, an "**a**dditive self-incrementing" buffer, and a "**m**ultiplicative self-incrementing" buffer. The buffer implementation is based on two associated data structures: a ***Buffer Descriptor*** (or *Buffer Handle)* and an array of characters (the actual character buffer). Both structures are to be created "on demand" at run time, that is, they are to be allocated dynamically. The ***Buffer Descriptor*** or ***Buffer Handle*** - the names suggest the purpose of this buffer control data structure - contains all the necessary information about the array of characters: a pointer to the beginning of the character array location in memory, the current capacity, the next character entry position, the increment factor, the operational mode and some additional parameters.

In this assignment you are to complete the coding for a "*buffer utility*", which includes the buffer data structure and the associated functions, following strictly the given specifications. Use the data declarations and function prototypes given below. **Do not change the names and the data types of the functions and the variables. Any change will be regarded as a serious specification violation.** Write the associated code.
The following structure declaration must be used to implement the Buffer Descriptor:

```
typedef struct BufferDescriptor {
    char *cb_head;      /* pointer to the beginning of character array (character buffer) */
    short capacity;     /* current dynamic memory size (in bytes) allocated to character buffer */
    short addc_offset;  /* the offset (in chars) to the add-character location */
    short getc_offset;  /* the offset (in chars) to the get-character location */
    short markc_offset; /* the offset (in chars) to the mark location */
    char  inc_factor;   /* character array increment factor*/
    char  mode;         /* operational mode indicator*/
    unsigned short  flags;  /* contains character array reallocation flag and end-of-buffer flag */
} Buffer, *pBuffer;
```

Where:

capacity is the current total size (measured in bytes) of the memory allocated for the character array by ***malloc()/realloc()*** functions. In the text below it is referred also as current capacity. It is whatever value you have used in the call to ***malloc()/realloc()*** that allocates the storage pointed to by ***cb_head***.

inc_factor is a buffer increment factor. It is used in the calculations of a new buffer **capacity** when the buffer needs to grow. The buffer needs to grow when it is full but still another character needs to be added to the buffer. The buffer is full when ***addc_offset*** measured in bytes is equal to ***capacity*** and thus all the allocated memory has been used. The ***inc_factor*** is only used when the buffer operates in one of the "self-incrementing" modes. In "**a**dditive self-incrementing" mode it is a positive integer number in the range of 1 to 255 and represents directly the increment (measured in characters) that must be added to the current capacity every time the buffer needs to grow. In "**m**ultiplicative self-incrementing" mode it is a positive integer number in the range of 1 to 100 and represents a percentage used to calculate the new capacity increment that must be added to the current capacity every time the buffer needs to grow.

addc_offset is the distance (measured in chars) from the beginning of the character array (***cb_head***) to the location where the next character is to be added to the existing buffer content. ***addc_offset*** (measured in bytes) must never be larger than ***capacity***, or else you are overrunning the buffer in memory and your program may crash at run-time or destroy data.

getc_offset is the distance (measured in chars) from the beginning of the character array (***cb_head***) to the location of the character which will be returned if the function ***b_getc()*** is called. The value ***getc_offset*** (measured in chars) must never be larger than ***addc_offset***, or else you are overrunning the buffer in memory and your program may get wrong data or crash at run-time. If the value of ***getc_offset*** is equal to the value of ***addc_offset***, the buffer has reached the end of its current content.

markc_offset is the distance (measured in chars) from the beginning of the character array (***cb_head***) to the location of a *mark*. A *mark* is a location in the buffer, which indicates the position of a specific character (for example, the beginning of a word or a phrase).

mode is an operational mode indicator. It can be set to three different integer numbers: 1, 0, and –1. The number **0** indicates that the buffer operates in "**f**ixed-size" mode; **1** indicates "**a**dditive self-incrementing" mode; and **–1** indicates "**m**ultiplicative self-incrementing" mode. The mode is set when a new buffer is created and cannot be changed later.

flags is a field containing different flags and indicators. In cases when storage space must be as small as possible, the common approach is to pack several data items into single variable; one common use is a set of single-bit or multiple-bit flags or indicators in applications like compiler buffers, file buffers, and database fields. The flags are usually manipulated through different bitwise operations using a set of "masks." Alternative technique is to use *bit-fields.* Using bit-fields allows individual fields to be manipulated in the same way as structure members are manipulated. Since almost everything about bit-fields is implementation dependent, this approach should be avoided if the portability is a concern. In this implementation, you are to use bitwise operations and masks (see *bitmask.c* example).

Each flag or indicator uses one or more bits of the ***flags*** field. The flags usually indicate that something happened during a routine operation (end of file, end of buffer, integer arithmetic sign overflow, and so on). Multiple-bit indicators can indicate, for example, the mode of the buffer (three different combinations – therefore 2-bits are needed). In this implementation the ***flags*** field has the following structure:

Bit	**MSB** 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0 **LSB**
Contents	1	1	1	1	1	1	1	1	1	1	1	1	1	1	x	x
Description	reserved for future use must be set to 1 and stay 1 all the time in this implementation														**r_flag**	**eob** flag

The LSB bit (bit 0) of the ***flags*** field is a single-bit *end-of-buffer* (***eob***) flag. The ***eob*** bit is by default **0**, and when set to 1, it indicates that the end of the buffer content has been reached during the buffer read operation (***b_getc()*** function). If ***eob*** is set to **1**, the function ***b_getc()*** should not be called before the ***getc_offset*** is reset by another operation.

Bit 1 of the ***flags*** field is a single-bit reallocation flag (***r_flag***). The ***r_flag*** bit is by default **0**, and when set to 1, it indicates that the location of the buffer character array in memory has been changed due to memory reallocation. This could happen when the buffer needs to expand or shrink. The flag can be used to avoid dangling pointers when pointers instead of offsets are used to access the information in the character buffer.

The rest of the bits are reserved for further use and must be set by default to **1** and they must not be changed by the bitwise operation manipulating bit 0 and bit 1.

You are to implement the following set of buffer utility functions (operations). Later they will be used by all other parts of the compiler when a temporary storage space is needed.
The **first implementation step** in all functions must be the validation (if possible and appropriate) of the function arguments. If an argument value is invalid, the function must return an appropriate failure indicator.

Buffer * b_allocate (short init_capacity,char inc_factor,char o_mode)
This function creates a new buffer in memory (on the program heap). The function
- tries to allocate memory for one ***Buffer*** structure using ***calloc()***;
- tries to allocates memory for one dynamic character buffer (character array) calling ***malloc()*** with the given initial capacity ***init_capacity***. The range of the parameter ***init_capacity*** must be between 0 and the ***MAXIMUM ALLOWED POSITIVE VALUE*** – *1* inclusive. The *maximum allowed positive value* is determined by the data type of the parameter ***init_capacity***. The pointer returned by ***malloc()*** is assigned to ***cb_head***;
- sets the buffer operational mode indicator ***mode*** and the Buffer structure increment factor ***inc_factor***. If the *o_mode* is the symbol ***f*** or *inc_factor* is **0,** the ***mode*** and the buffer ***inc_factor*** are set to number **0**. If the *o_mode* is the symbol ***f*** and *inc_factor* is not **0,** the ***mode*** and the buffer ***inc_factor*** are set to 0. If the *o_mode* is ***a*** and *inc_factor* is in the range of **1** to **255** inclusive , the ***mode*** is set to number **1** and the buffer ***inc_factor*** is set to the value of *inc_factor*. If the *o_mode* is ***m*** and *inc_factor* is in the range of **1** to **100** inclusive, the ***mode*** is set to number **-1** and the *inc_factor* value is assigned to the buffer ***inc_factor***;
- copies the given ***init_capacity*** value into the Buffer structure ***capacity*** variable;
- sets the ***flags*** field to its default value which is FFFC hexadecimal.

Finally, on success, the function returns a pointer to the ***Buffer*** structure. It must return **NULL** pointer on any error which violates the constraints imposed upon the buffer parameters or prevents the creation of a working buffer. If run-time error occurs, the function must return immediately after the error is discovered. Check for all possible errors which can occur at run time. Do not allow **"memory leaks", "dangling"** pointers, or "**bad**" parameters.

pBuffer b_addc (pBuffer const pBD, char symbol)
Using a bitwise operation the function resets the ***flags*** field ***r_flag*** bit to 0 and tries to add the character ***symbol*** to the character array of the given ***buffer*** pointed by ***pBD***. If the buffer is operational and it is not full, the symbol can be stored in the character buffer. In this case, the function adds the character to the content of the character buffer, increments ***addc_offset*** by 1 and returns.

If the character buffer is already full, the function will try to resize the buffer by increasing the current capacity to a new capacity. How the capacity is increased depends on the current operational mode of the buffer.

If the operational mode is **0**, the function returns NULL.

If the operational mode is **1**, it tries to increase the current capacity of the buffer to a *new capacity* by adding ***inc_factor*** (converted to bytes) to ***capacity***. If the result from the operation is positive and does not exceed the ***MAXIMUM ALLOWED POSITIVE VALUE*** *–1 (*minus *1)*, the function proceeds. If the result from the operation is positive but exceeds the ***MAXIMUM ALLOWED POSITIVE VALUE*** *–1 (*minus *1)*, it assigns the ***MAXIMUM ALLOWED POSITIVE VALUE*** *–1* to the *new capacity* and proceeds. The ***MAXIMUM ALLOWED POSITIVE VALUE*** is determined by the data type of the variable, which contains the buffer capacity.
If the result from the operation is negative, it returns NULL.

If the operational mode is **-1** it tries to increase the current capacity of the buffer to a *new capacity* in the following manner:

- If the current capacity can not be incremented anymore because it has already reached the maximum capacity of the buffer, the function returns NULL.

The function tries to increase the current capacity using the following formulae:

available space = maximum buffer capacity – current capacity
*new increment = available space * inc_factor / 100*
new capacity = current capacity + new increment

The *maximum buffer capacity* is the ***MAXIMUM ALLOWED POSITIVE VALUE*** *–1.* If the *new capacity* has been incremented successfully, no further adjustment of the *new capacity* is required.
If as a result of the calculations, the *current capacity* cannot be incremented, but the *current capacity* is still smaller than the ***MAXIMUM ALLOWED POSITIVE VALUE*** *–1*, then the ***MAXIMUM ALLOWED POSITIVE VALUE*** *–1* is assigned to the *new capacity* and the function proceeds***.***

If the capacity increment in mode **1** or **-1** is successful, the function performs the following operations:

- the function tries to expand the character buffer calling ***realloc()*** with the *new capacity*. If the reallocation fails, the function returns NULL;
- if the location in memory of the character buffer has been changed by the reallocation, the function sets ***r_flag*** bit to **1** using a bitwise operation;
- adds (appends) the character ***symbol*** to the buffer content;
- changes the value of ***addc_offset*** by 1, and saves the newly calculated capacity value into ***capacity*** variable;
- the function returns a pointer to the **Buffer** structure.

The function must return NULL on any error. Some of the possible errors are indicated above but you must check for all possible errors that can occur at run-time. Do not allow "**memory leaks**". Avoid creating "**dangling pointers**" and using "**bad**" parameters. The function ***must not destroy*** the buffer or the contents of the buffer even when an error occurs – it must simply return NULL leaving the existing buffer content intact. **A change in the project platform (16-bit, 32-bit or 64-bit) must not lead to improper behavior**.

int b_clear (Buffer * const pBD)
The function retains the memory space currently allocated to the buffer, but re-initializes all appropriate data members of the given ***Buffer*** structure (buffer descriptor), such that the buffer will appear empty and the next call to ***b_addc()*** will put the character at the beginning of the character buffer. The function does not need to clear the existing contents of the character buffer. If a run-time error is possible, the function should return **–1** in order to notify the calling function about the failure.

void b_free (Buffer * const pBD)
The function de-allocates (frees) the memory occupied by the character buffer and the ***Buffer*** structure (buffer descriptor). The function should not cause abnormal behavior (crash).

int b_isfull (Buffer * const pBD)
The function returns ***1*** if the character buffer is full; it returns ***0*** otherwise. If a run-time error is possible, the function should return **–1**.

short b_limit (Buffer * const pBD)
The function returns the current limit of the character buffer. The current limit is the amount of space measured in chars that is currently being used by all added (stored) characters. If a run-time error is possible, the function should return **–1**.

short b_capacity(Buffer * const pBD)
The function returns the current capacity of the character buffer. If a run-time error is possible, the function should return **–1**.

short b_mark (pBuffer const pBD, short mark)
The function sets ***markc_offset*** to ***mark***. The parameter ***mark*** must be within the current limit of the buffer (0 to ***addc_offset*** inclusive). The function returns the currently set ***markc_offset***. If a run-time error is possible, the function should return **-1.**

int b_mode (Buffer * const pBD)
The function returns the value of ***mode*** to the calling function. If a run-time error is possible, the function should notify the calling function about the failure .

size_t b_incfactor (Buffer * const pBD)
The function returns the non-negative value of ***inc_factor*** to the calling function. If a run-time error is possible, the function should return **0x100**.

int b_load (FILE * const fi, Buffer * const pBD)
The function loads (reads) an open input file specified by ***fi*** into a buffer specified by ***pBD***. The function must use the standard function ***fgetc(fi)*** to read one character at a time and the function ***b_addc()*** to add the character to the buffer. If the current character cannot be added to the buffer, the function returns the character to the file stream (file buffer) using ***ungetc()*** library function, then prints the returned character both as a character and as an integer (see test file ass1fi.out) and then returns **-2** (use the defined *LOAD_FAIL* constant). The operation is repeated until the standard macro ***feof(fi)*** detects end-of-file on the input file. The end-of-file character must not be added to the content of the buffer.
Only the standard macro ***feof(fi)*** must be used to detect end-of-file on the input file. Using other means to detect end-of-file on the input file will be considered a significant specification violation.
If some other run-time errors are possible, the function should return **–1**. If the loading operation is successful, the function must return the number of characters added to the buffer.

int b_isempty (Buffer * const pBD)
If the ***addc_offset*** is 0, the function returns 1; otherwise it returns 0. If a run-time error is possible, it should return **–1**.

char b_getc (Buffer * const pBD)
This function is used to read the buffer. The function performs the following steps:
- checks the argument for validity (possible run-time error). If it is not valid, it returns **-2**;
- if ***getc_offset*** and ***addc_offset*** are equal, using a bitwise operation it sets the ***flags*** field ***eob*** bit to **1** and returns number **0**; otherwise, using a bitwise operation it sets ***eob*** to **0**;
- returns the character located at ***getc_offset***. Before returning it increments ***getc_offset*** by **1**.

int b_eob (Buffer * const pBD)
The function returns the ***eob*** bit value to the calling function. A bitwise operation must be used to return the value of the ***flags*** field ***eob*** bit. If a run-time error is possible, it should return **–1**.

int b_print (Buffer * const pBD)
The function is intended to be used for used for diagnostic purposes only. Using the ***printf()*** library function the function prints character by character the contents of the character buffer to the standard output (stdout). Before printing the content the function checks if the buffer is empty, and if it is, it prints the following message `Empty buffer!` adding a new line at the end and returns. Next, the function prints the content calling ***b_getc()*** in a loop and using ***b_eob()*** to detect the end of the buffer content. Finally, it prints a new line character. It returns the number of characters printed. The function returns **–1** on failure.

Buffer * b_compact(Buffer * const pBD, char symbol)
For all operational modes of the buffer the function shrinks (or in some cases may expand) the buffer to a *new capacity*. The *new capacity* is the current limit plus a space for one more character. In other words the *new capacity* is ***addc_offset + 1*** converted to bytes. The function uses ***realloc()*** to adjust the *new capacity*, and then updates all the necessary members of the buffer descriptor structure. Before returning a pointer to ***Buffer***, the function adds the ***symbol*** to the end of the character buffer (**do not** use ***b_addc()***, use ***addc_offset*** to add the symbol) and increments

addc_offset. The function must return NULL if for some reason it cannot to perform the required operation. It must set the ***r_flag*** bit appropriately***.***

char b_rflag (Buffer * const pBD)
The function returns the ***r_flag*** bit value to the calling function. A bitwise operation must be used to return the value of the ***flags*** field ***r_flag*** bit. If a run-time error is possible, it should return **–1**.

short b_retract (Buffer * const pBD)
The function decrements ***getc_offset*** by **1**. If a run-time error is possible, it should return **–1**; otherwise it returns ***getc_offset***.

short b_reset (Buffer * const pBD)
The function sets ***getc_offset*** to the value of the current ***markc_offset*** . If a run-time error is possible, it should return **–1**; otherwise it returns ***getc_offset***.

short b_getcoffset (Buffer * const pBD)
The function returns ***getc_offset*** to the calling function. If a run-time error is possible, it should return **–1**.

int b_rewind(Buffer * const pBD)
The function set the ***getc_offset*** and ***markc_offset*** to 0, so that the buffer can be reread again. If a run-time error is possible, it should return **–1**; otherwise it returns 0;

char * b_location(Buffer * const pBD, short loc_offset)
The function returns a pointer to a location of the character buffer indicated by ***loc_offset***. ***loc_offset*** is the distance (measured in chars) from the beginning of the character array (***cb_head***). If a run-time error is possible, it should return **NULL**.

All constant definitions, data type and function declarations (prototypes) must be located in a header file named ***buffer.h***. You are allowed to use only named constants in your programs (except when incrementing something by 1 or setting a numeric value to 0). To name a constant you must use ***#define*** preprocessor directive (see ***buffer.h***). The incomplete ***buffer.h*** is posted on Brightspace (BS). The function definitions must be stored in a file named ***buffer.c***.

<u>Task 2:</u> Testing the Buffer

To test your program you are to use the test harness program ***platy_bt.c*** (do not modify it) and the input files ***ass1e.pls*** (an empty file), and ***ass1.pls***. The corresponding output files are ***ass1e.out*** and ***ass1ai.out*** (mode = 1), ***ass1mi.out*** (mode = -1), ***ass1fi.out*** (mode = 0). Those files are available as part of the assignment postings. You must create a standard console project named ***buffer*** with an executable target ***buffer*** (see ***Creating_C_Project*** document in Lab0). The project must contain only one header file (***buffer.h***) and two souce files: ***buffer.c*** and ***platy_bt.c***.

Here is a brief description of the program that is provided for you on Brightspace (BS). It simulates “normal” operating conditions for your buffer utility. The program (***platy_bt.c***) main function takes to parameters from the command line: an input file name and a character (**f** - fixed-size, **a** – additive self-increment, or **m** – multiplicative self increment) specifying the buffer operational mode. It opens up a file with the specified name (for example, ***ass1.pls***), creates a buffer, and loads it with data from the file using the ***b_load()*** function. Then the program prints the current capacity, the current

limit, the current operational mode, the increment factor, the current mark, and the contents of the buffer. It packs the buffer, and if the pack operation is successful, it prints the buffer contents again. Your program must not overflow any buffers in any operational mode, no matter how long the input file is. The provided main program will not test all your functions. You are strongly encouraged to test all your buffer functions with your own test files and modified main function.

Bonus Task: **Implementing a Preprocessor Macro Definition and Expansion (1%)**

Implement ***b_isfull()*** both as a function and a macro expansion (macro). Using conditional processing you must allow the user to choose between using the macro or the function in the compiled code. If **B_FULL** name is defined the macro should be used in the compiled code. If the **B_FULL** name is not defined or undefined, the function should be used in the compiled code. To receive credit for the bonus task your code must be well documented, tested, and working.

SUBMIT THE FOLLOWING:

Paper Submission: Hand in on paper, the fully documented source code of your ***buffer.h*** and ***buffer.c*** files. Print your output for the test files ***ass1.pls*** and ***ass1e.pls***. Include a description or listing of your own test file(s) (output and/or input, if appropriate) showing how you tested your program. Don't kill a hundred trees; submit descriptions and short excerpts of your testing inputs and outputs if the files are large. Save the Third Rock from the Sun! ***Print and submit the Marking Sheet for Assignment 1 with your Name and Student ID filled in.***

Digital Submission: Compress into a **zip** file the following files*: platy_bt.c, buffer.h, buffer.c, ass1.pls, ass1e.pls* and the corresponding *test output files* produced by your program. Include your additional input/output test files if you have any. Upload the zip file on Brightspace. The file must be submitted prior or on the due date as indicated in the assignment. The name of the file must be **Your Last Name** followed by the last three digits of your student number. For example: Ranev345.zip.

Make sure all printed materials are placed into an unsealed envelope and are deposited into my assignment submission box prior to the end of the due date. If the due time is midnight, you can make the submission the next morning. The submission must follow the course submission standards. You will find the Assignment Submission Standard as well as the Assignment Marking Guide (**CST8152_ASSAMG.pdf)** for the Compilers course on the Brightspace.

Assignments will not be marked if the source files are not submitted on time. Assignments could be late, but the lateness will affect negatively your mark: see the Course Outline and the Marking Guide. All assignments must be successfully completed to receive credit for the course, even if the assignments are late.

Evaluation Note: Make your functions as efficient as possible. These functions are called many times during the compilation process. The functions will be graded with respect to design, documentation, error checking, robustness, and efficiency. When evaluating and marking your assignment, I will use the standard project and ***platy_bt.c*** and the test files posted on the net. If your program compiles, runs, and produces correct output files, it will be considered a *working program*. Additionally, I will try my best to "crash" your functions using a modified main program, which will test all your functions including calling them with "invalid" parameters. I will use also some additional test files (for example, a large file). This can lead to fairly big reduction of your assignment mark (see **CST8152_ASSAMG** and ***c*MarkingSheetA1** documents).

Enjoy the assignment. And do not forget that:

"Writing a program is like painting. It is better to start on a new canvas." Ancient P-Artist

"It is part of the nature of humans to begin with romance (buffer) and build to reality (compiler)." by Ray Bradbury

#define buff·er (bŭf′ər) *noun (Microsoft Bookshelf)*

1. Something that lessens or absorbs the shock of an impact.
2. One that protects by intercepting or moderating adverse pressures or influences: *"A sense of humor . . . may have served as a buffer against the . . . shocks of disappointment"* (James Russell Lowell).
3. Something that separates potentially antagonistic entities, as an area between two rival powers that serves to lessen the danger of conflict.
4. *Chemistry*. A substance that minimizes change in the acidity of a solution when an acid or base is added to the solution.
5. ***Computer Science*. A device or memory area used to store data temporarily and deliver it at a rate different from that at which it was received.**

S^R, CST8152 – Compilers, September, 2018