ALGONQUIN COLLEGE

ASSIGNMENT 1 SPECIFICATION[1] - BUFFER

General View

Due Date: prior or on **October 3rd 2020 (midnight)**

- **1st week late submission** (until October 10th midnight): 30% off.
- **2nd week late submission** (until October 17th midnight): 50% off.
- **Later:** 100% off.

Earnings: 5% 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:
- The current version of code requires *Camel Code style*. Use it appropriately.

[1] Adapted from resources developed by Prof. Svillen Ranev (Algonquin College, 2019)

Task 1: Buffer implementation

1.1. BUFFER STRUCTURE

Note 1: Remembering Buffers

Buffers *are often used when developing compilers because of their efficiency (see page 111 of your textbook).*

The buffer implementation is based on two associated data structures: a Buffer entity (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.

- You are to implement a buffer that can operate in three different modes: a "**fixed-size**" buffer, an "**additive self-incrementing**" buffer, and a "**multiplicative self-incrementing**" buffer.

The following structure declaration must be used to implement the Buffer Entity:

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

Where:

- ***string*** is the pointer that indicates the beginning of useful information (loaded from a source file).

- ***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 ***string***.
- ***increment*** 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 ***addCPosition*** measured in bytes is equal to ***capacity*** and thus all the allocated memory has been used. The ***increment*** 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.
    - ***NOTE:*** As expected, in "**f**ixed" mode, once defined the size, the **buffer size cannot increase**.
- ***addCPosition*** is the distance (measured in chars) from the beginning of the character array (***string***) to the location where the next character is to be added to the existing buffer content.
    - ***NOTE: addCPosition*** 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.
- ***getCPosition*** is the distance (measured in chars) from the beginning of the character array (***string***) to the location of the character which will be returned if the function ***bufferGetChar()*** is called.
    - ***NOTE:*** The value ***getCPosition*** must never be larger than ***addCPosition***, 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 ***getCPosition*** is equal to the value of ***addCPosition***, the buffer has reached the end of its current content.
- ***markCPosition*** is the distance (measured in chars) from the beginning of the character array (***string***) to the location of a ***mark***.
    - ***NOTE:*** 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). It will be used in the Scanner, but the implementation must be done in buffer.
- ***opMode*** is an operational mode indicator. It can be set to three different integer numbers: 1, 0, and -1. The mode is set when a new buffer is created and cannot be changed later.
    - The number **0** indicates that the buffer operates in "**f**ixed-size" mode

- Number **1** indicates "**a**dditive self-incrementing" mode
- And number **-1** indicates "**m**ultiplicative self-incrementing" mode.

❖ ***flags*** is a field containing different flags and indicators.

> ***Note 2: Remembering Flags***
>
> *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.*

- 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**
Content	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

- Note that the sequence of 111…100 (2 bytes) correspond to the value **FFFC** hexadecimal (see **buffer.h**).
- The LSB 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 (***bufferGetChar()*** function). If ***eob*** is set to **1**, the function ***bufferGetChar()*** should not be called before the ***getCPosition*** is reset by another operation.
- The LSB 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 (positions) are used to access the information in the character buffer.

- ***NOTE:*** The rest of the bits are resetting ***eob*** or ***r_flag*** reserved for further use and **must be set by default to 1**. When setting or they must not be changed by the bitwise operation manipulating bit 0 and 1.

❖ 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.

- ***IMPORTANT NOTE:*** **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.**

1.2. BUFFER FUNCTIONALITIES

You are to implement the following set of buffer utility functions (operations). All function definitions must be stored in a file named ***buffer.c***. 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 immediately an appropriate failure indicator.

Note 3: Programming Best Practices
In Compilers, you are invited to show your best practices. Some of they are already illustrated here:
- Standard for codification (here, we are using Camel syntax to make code better readable. Especially when we are not using OO paradigm, the construction of methods and variables should be understood by anyone.
- Boundary conditions: all codes should consider problem (normally during their initialization = parameter checking). See several "Error conditions" shown below.
*- Use **DEFENSIVE PROGRAMMING**: Test not only the initial conditions, but rules from specification carefully.*

❖ ***pBuffer bufferCreate (short initCapacity ,char incFactor ,char opMode)***

- This function creates a new buffer in memory (on the program heap).
- The function tries to allocate memory for one ***Buffer*** structure using ***calloc()***;
- It tries to allocates memory for one dynamic character buffer (character array) calling ***malloc()*** with the given initial capacity ***initCapacity***.

- ***NOTE:*** The range of the parameter ***initCapacity*** 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 ***initCapacity***.
  - If the ***initCapacity*** is **0,** the function tries to create a character buffer with default size **200** characters.
  - If the ***initCapacity*** is **0,** the function ignores the current value of the parameter ***initCapacity*** and sets the buffer field ***increment*** to **15** ***(DEFAULT_INC_FACTOR)*** in mode ***a*** and ***m*** or to **0** in mode ***f***.
  - The pointer returned by ***malloc()*** is assigned to ***string***;
- This function also sets the Buffer structure operational mode indicator ***opMode*** and the ***increment***.
  - If the *opMode* is the symbol ***f*****,** the ***mode*** and the buffer ***increment*** are set to number **0**.
  - If the *incFactor* parameter is **0** and *initCapacity* is not **0** (see above)**,** the ***opMode*** and the buffer ***increment*** are set to **0**.
  - If the *opMode* is ***a*** and *incFactor* is in the range of **1** to **255** inclusive, the ***opMode*** is set to number **1** and the buffer ***increment*** is set to the value of *incFactor*.
  - If the *opMode* is ***m*** and *incFactor* is in the range of **1** to **100** inclusive, the ***opMode*** is set to number **-1** and the *incFactor* value is assigned to the buffer ***increment***;
- This function also copies the given *initCapacity* value into the Buffer structure ***capacity*** variable;
- It also 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.
- ***Error Condition:*** 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 bufferAddChar (pBuffer const pBE, 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 ***pBE***.

- ***NOTE:*** 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 ***addCPosition*** by 1 and returns.
- ***NOTE:*** 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 ***increment*** (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.
    - ***Error Condition:*** 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
```

    - **TIP**: To use this formula, check eventual casting to guarantee acceptable values.
    - 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.
    - ***NOTE:*** 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 *–1* is assigned to the *new capacity* and the function proceeds.

- If the capacity increment in mode **1 ('a')** or **-1** ('**m**') is successful, the function performs the following operations:
  - The function tries to expand the character buffer calling ***realloc()*** with the *new capacity*.
  - 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;
  - In short, it adds (appends) the character ***symbol*** to the buffer content;
  - It also changes the value of ***addCPosition*** by 1, and saves the newly calculated capacity value into ***capacity*** variable;
  - In the end, the function returns a pointer to the **Buffer** structure.
- ***Error Condition:*** **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.
  - **NOTE:** A change in the project platform (16-bit, 32-bit or 64-bit) must not lead to improper behavior.

❖ ***int bufferClear (Buffer * const pBE)***

- 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) so that the buffer will appear as just created to the client functions (for example, next call to ***bufferAddChar()*** 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.
- ***Error Condition:*** If a run-time error is possible, the function should return **–1** in order to notify the calling function about the failure.

❖ ***void bufferFree (Buffer * const pBE)***

- The function de-allocates (frees) the memory occupied by the character buffer and the ***Buffer*** structure (buffer descriptor).
- ***Error Condition:*** The function should not cause abnormal behavior (crash).

❖ ***int bufferIsFull (Buffer * const pBE)***

- The function returns ***1*** if the character buffer is full; it returns ***0*** otherwise.
- ***Error Condition:*** If a run-time error is possible, the function should return **-1**.

❖ ***short bufferGetAddCPosition (Buffer * const pBE)***

- The function returns the current ***addCPosition***.
- ***Error Condition:*** If a run-time error is possible, the function should return **-1**.

❖ ***short bufferGetCapacity (Buffer * const pBE)***

- The function returns the current capacity of the character buffer.
- ***Error Condition:*** If a run-time error is possible, the function should return **-1**.

❖ ***int bufferGetOpMode (Buffer * const pBE)***

- The function returns the value of ***opMode*** to the calling function.
- ***Error Condition:*** If a run-time error is possible, the function should notify the calling function about the failure.

❖ ***size_t bufferGetIncrement (Buffer * const pBE)***

- The function returns the non-negative value of ***increment*** to the calling function. The return datatype (size_t) must be appropriately checked according to datatype you are getting.
- ***Error Condition:*** If a run-time error is possible, the function should return a special value: **0x100**.

❖ ***int bufferLoad (FILE * const fi, Buffer * const pBE)***

- The function loads (reads) an open input file specified by ***fi*** into a buffer specified by ***pBE***. The file is supposed to be a plain file (for instance, a PLATYPUS code).

- **TIP**: The function must use the standard function ***fgetc(fi)*** to read one character at a time and the function ***bufferAddChar()*** to add the character to the buffer.
- ***NOTE:*** If the current character cannot be added to the buffer (***bufferAddChar()*** returns NULL), the function returns the character to the file stream (file buffer) using ***ungetc()*** library function 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.
- ***Error Condition:*** 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 bufferIsEmpty (Buffer * const pBE)***

- If the ***addCPosition*** is 0, the function returns 1; otherwise it returns 0.
- ***Error Condition:*** If a run-time error is possible, it should return **-1**.

❖ ***char bufferGetChar (Buffer * const pBE)***

- This function is used to read the buffer. The function performs the following steps:
  - If ***getCPosition*** and ***addCPosition*** 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**;
  - ***Error Condition:*** Checks the argument for validity (possible run-time error). If it is not valid, it returns **-2**.
- In the end, it increments ***getCPosition*** by **1** and returns the character located at ***getCPosition***.

❖ ***int bufferPrint (Buffer * const pBE, char newLine)***

- This function is intended to print the content of buffer (in ***string*** field).
- Using the ***printf()*** library function the function prints character by character the contents of the character buffer to the standard output (stdout).

- In a loop the function prints the content of the buffer calling ***bufferGetChar()*** and **checking the flags** in order to detect the **end of the buffer** content (using other means to detect the end of buffer content will be considered a significant specification violation).
- After the loop ends, it checks the ***newLine*** and if it is not **0,** it prints a new line character.
- Finally, it returns the number of characters printed.
- ***Error Condition:*** The function returns **-1** on failure.

❖ ***Buffer* bufferSetEnd (Buffer * const pBE, 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 ***getCPosition + 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 ***bufferAddChar()***, use ***addCPosition*** to add the symbol) and increments ***addCPosition.***
- ***NOTE:*** It must set the ***r_flag*** bit appropriately***.***
- ***Error Condition:*** The function must return NULL if for some reason it cannot to perform the required operation.

❖ ***short bufferRetract (Buffer * const pBE)***

- The function decrements ***getCPosition*** by **1**.
- ***Error Condition:*** If a run-time error is possible, it should return **-1**; otherwise it returns ***getCPosition***.

❖ ***short bufferReset (Buffer * const pBE)***

- The function sets ***getCPosition*** to the value of the current ***markCPosition***.
- ***Error Condition:*** If a run-time error is possible, it should return **-1**; otherwise it returns ***getCPosition***.

❖ ***short bufferGetCPosition (Buffer * const pBE)***

- The function returns ***getCPosition*** to the calling function.
- ***Error Condition:*** If a run-time error is possible, it should return **-1**.

❖ ***int bufferRewind (Buffer * const pBE)***

- The function set the ***getCPosition*** and ***markCPosition*** to 0, so that the buffer can be reread again.
- ***Error Condition:*** If a run-time error is possible, it should return **-1**; otherwise it returns 0.

❖ ***char * bufferGetString (Buffer * const pBE, short charPosition)***

- The function returns a pointer to the location of the character buffer indicated by ***charPosition*** that is the distance (measured in chars) from the beginning of the character array (***string***).
- ***Error Condition:*** If a run-time error is possible, it should return **NULL**.

❖ ***pBuffer bufferSetMarkPosition (pBuffer const pBE, short mark)***

- The function sets ***markCPosition*** to ***mark***.
- The parameter ***mark*** must be within the current limit of the buffer (0 to ***addCPosition*** inclusive).
- The function returns the currently set ***markCPosition***.
- ***Error Condition:*** If a run-time error is possible, the function should return.

❖ ***unsigned short bufferGetFlags (pBuffer const pBE)***

- The function returns the flag field from buffer;
- ***Error Condition:*** If a run-time error is possible, it should return an error code (**-1** on failure).

1.3. BUFFER HEADER FILE

❖ 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***).
- ***IMPORTANT NOTE:*** The incomplete ***buffer.h*** is posted on Brightspace (BS). All function definitions must be stored in a file named ***buffer.c***.

Task 2: Buffer Test

2.1. GENERAL VIEW

To test your program, you are to use the test harness program ***testBuffer.c*** **(do not modify it)** and test it with input files.

- The input files ***a1e.pls*** (an empty file), and ***a1r.pls*** (a correct PLATYPUS file) will be provided.
- The corresponding output files are:
  - When processing a1e.pls:
    - ***a1e.out***: For empty file (using "a" mode);
  - When processing a1r.pls:
    - ***a1fi.out*** (for mode = 0). ***a1ai.out*** (for mode = 1), ***a1mi.out*** (for mode = -1).
- Those files are generated by my test program and contain plain ASCII text.
- They are available as part of the assignment postings on Brightspace (BS).
- ***IMPORTANT NOTE:*** You must create a standard console project named ***buffer.exe*** with an executable target ***buffer*** (see ***Creating_C_Project*** document in Lab0).
- The project must contain only one header file (***buffer.h***) and one C file: ***buffer.c***.

TIP:

There is a very useful program you should download and install on your computer: **Total Commander**. This is very useful file and program manager (Window Explorer is supposed to be such a program). It is also a very powerful FTP client and has built-in zip/unzip utilities. And it could be very helpful when you need to compare

2.2. ABOUT EXECUTION

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 (***testBuffer.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 (calling ***bufferCreate()***), and loads it with data from the file using the ***bufferLoad()*** function.
  - Then the program prints the current capacity (***bufferGetCapacity()***), the current size (***bufferGetAddCPosition()***), the current operational mode (***bufferGetOpMode()***), the increment factor (***bufferGetIncrement()***), the first symbol (using ***bufferGetString()***), the flags (***bufferGetFlags()***), and rewinds (***bufferRewind()***) in order to print the contents of the buffer (calling ***bufferPrint()***).
  - Finally, it compacts the buffer (***bufferSetEnd()***), and if the operation is successful, it prints the buffer contents again and frees the buffer (***bufferFree()***).
- ***NOTE:*** Your program must not overflow any buffers in any operational mode, no matter how long the input file is.
- **TIP**: **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.

Task 3: Bonus Task

You have a chance to get bonus by **implementing a Preprocessor Macro Definition and Expansion (1%)**

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

Submission Details

- **Digital Submission: Compress** into a **zip** file the following files: **buffer.h**, **buffer.c** and additional files related to submission - your additional input/output test files if you

have any. Also include a Cover page and a Test Plan. Check the A1 Marking Sheet for it, as well as Submission Standard document.

- 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.
- Upload the zip file on Brightspace. The file must be submitted prior or on the due date as indicated in the assignment.
- ***IMPORTANT NOTE:*** The name of the file must be **Your Last Name** followed by the last three digits of your student number followed by your lab **section number**. For example: **Sousa123_s10.zip**.
  - If you are working in teams, please, include also your partner. For instance, something like: **Sousa123_Melo456_s10.zip**.
  - **Remember:** Only students from the same section can constitute a specific team.
- ***IMPORTANT NOTE:*** **Assignments will not be marked if there are not source files in the digital submission.** 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 **testBuffer.c** and the test files posted on Brightspace.
  - 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). So, test your code as much as you can!
  - This can lead to fairly big reduction of your assignment mark (see **CST8152_ASSAMG** and **MarkingSheetA1** documents).

About Lab Demo

- **Main Idea:** This semester, you can get **bonuses** when you are demonstrating your evolution in labs. The marks are reported in CSI.

- **How to Proceed:** You need to demonstrate the expected portion of code to your Lab Professor in **private Zoom Sections**.
  - If you are working in teams, you and your partner must do it together, otherwise, only the student that has presented can get the bonus marks.
  - Eventual questions can be posed by the Lab professor for any explanation about the code developed.
  - Each demo is related to a **specific lab** in one specific week. If it is not presented, no marks will be given later (even if the activity has been done).

Finally, as professor **Svillen Ranev** used to say, **enjoy the assignment!** Yes, it is possible! And do not forget this message:

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

File update: Sep 18th 2020 by Paulo Sousa.

- *Changes:*
  - Better logic definition and corrections.
  - No more ***bufferGetEobFlag()*** and ***bufferGetRFlag()*** functions – including references in ***buffferPrint()***.
  - Macro function changed (to use only ***bufferGetFlags()***).

Good Luck with Assignment 1!