ALGONQUIN COLLEGE

ASSIGNMENT 1.2 – NEW READER IMPLEMENTATION

General View

Due Date: prior or on **Oct 2nd 2022 (midnight)**

- **2nd Due date** (until Oct 9th) - 50% off.

Earnings: 5% of your course grade

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

- This is the second new task in Compilers: You need to create your Reader to be used in your new language. Follow the example of **BOA** implementation.
- This is a review of and an exercise in:
  - C coding style (ex: pre-compilation);
  - Data types and structures;
  - Memory management (pointers);
  - File input/output;
  - Programming technique (including defensive programming).

Note 1: The Defensive Programming
Specially in C programs, where you can manipulate pointers and the datatypes can be implicitly converted and also errors can give you unexpected results, you need to be responsible for what you are doing. **Examples**: suppose that you receive a reader pointer as parameter. How to be sure that in fact there is a valid address? Think about positions in an array? What are the boundary conditions? And when you adding positive values, what are the risks that can affect your result?

- To be sure that you covered all required activities, please check the TODO notes in this specification.
- It will give you a better understanding of the type of the **reader** as the internal data structure used by a simple compiler to be developed. This assignment will be also an exercise in "*excessively defensive programming*".
- You are to write / complete 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:

- **Code Style** (recommendation): The current version of code requires *Camel Code style* (ex: "bufCreate"). The objective is let you to create not only a "beautiful code" but an "understandable" one.

Reader (before start coding)

Here are some tips (not exactly in a "logical" sequence of steps), with some ideas to help you during the development of A12 (**Reader**):

- Please read the Assignment **Submission Standard and Marking Guide** (at "Assignments > Standards" section).
- To do this activity:
  - You will see that you need to:
    - Download the files (see the **ZIP** file at A12 Assignment).
    - In your Project, import the .C and .H files in (including **Compilers.h, Compilers.c, Reader.h, Reader.c** and **MainReader.c**).
      - In short: Compilers.c uses definitions from Compilers.h and invokes MainReader.c that creates the data structure defined in Reader.h by Reader.c functions.
    - You can also see all **input files** (and scripts – **batch** files) used by professor in the **BOA** code.

Note 2: Input Files
The input files are mandatory – remember the files used are specific for each language. In the case of the professor demo, **BOA** files (extension: **.boa**) are provided. These files are plain text files (remember: you need to use ASCII symbols only)[1].

Reader implementation

- You will see a partial code developed to **BOA language** ("**Reader.h**" and "**Reader.c**"), check all **TODO** comments.

1.1. COMPILER – Header file

- In this file (**Compiler.h**), you need to define the main elements used in all assignments and basic definitions are used for **Reader**, **Scanner** and **Parser**.
- The most important part is to use typedefs to define your own language datatypes.

[1] Remember that in the **A11**, you defined the proposal of your language: name, extension and, most important, the basic syntax.

ACTION
Ex: If your language is called "Brazil", then, your datatypes are supposed to be something like "br_ch" (or similar) etc.

❖ Before start, remember that the **reader** is using datatypes related to the language. It is done by typedef definitions linking ANSI C datatypes with the language (ex: **BOA**) datatypes. Check the *header* about these definitions (that you can continue):

```
typedef char              boa_char;
typedef int               boa_intg;
typedef float             boa_real;
typedef void              boa_void;

typedef unsigned char     boa_boln;
typedef unsigned char     boa_byte;

typedef long              boa_long;
typedef double            boa_doub;
```

Note 3: About Datatypes

You need to define not only elements that you will use in your language (minimally, a numeric datatype - it can be an int or a float point and the string datatype), but others that are necessary to functions. Your language must match the datatypes that you are defining with some real datatypes that C compiler can translate. Ex: my "*br_ch*" is supposed to match with char C datatype. See the examples shown in **Compilers.h**.

1.2. COMPILER – Source code

❖ In **Compiler.c**, you will see the main method of the project. Essentially, according to inputs, specific functions will be called: (Ex: "1" will invoke "mainReader()", etc.).

➢ **TIP**: You just need to comment the code to invoke the appropriate function.

1.3. READER STRUCTURES – Header file

Note 4: About Readers

Readers / Buffers are often used when developing compilers because of their efficiency.

The **reader** implementation is based on data structures: The **Reader** (and this data structure uses **position**). The **reader** will be created "*on demand*" at run time, that is, they are to be allocated dynamically.

The **reader** is used to control data the main structure (the content of the **reader** – remember that in C, we must use **pointer to chars**) – that contains all the necessary information about the array of characters.

We need also to consider some elements in the **reader** data structure: a pointer to the beginning of the character array location in memory, the current size, the next character entry position, the increment factor, the operational mode and some additional parameters.

ACTION
Use your datatypes (defined in Compilers.h) to define all variables using your language prefix.

- In the code, you have a generic definition (see the **enum** READER_MODE and the READER_ERROR), that should not be changed.
- However, several constants are included and must be adjusted according to your specification. So: Use your language name and, if required, change the values (this is optional).
    - For instance, when you see "READER_MAX_SIZE" and other constants, you can change them or not.

- TODO: *Define the alias to your language – obviously, you need to adapt your original datatype to an ANSI C Datatype*.

The following structure declaration must be used to implement the **Reader**:

```
/* Reader structure and pointer */
typedef struct bufferReader {
        boa_char* content;              /* pointer to the beginning of character array (character Reader) */
        boa_intg size;                  /* current dynamic memory size (in bytes) allocated to character Reader */
        boa_intg increment;             /* character array increment factor */
        boa_char mode;                  /* operational mode indicator */
        boa_byte flags;                 /* contains character array reallocation flag and end-of-reader flag */
        boa_intg histogram[NCHAR];      /* contains character array reallocation flag and end-of-reader flag */
        Position position;
} BufferReader, *ReaderPointer;
```

And:

```
/* Offset information */
typedef struct position {
        boa_int mark;     /* the offset (in chars) to the add-character location */
        boa_int read;    /* the offset (in chars) to the get-character location */
        boa_int wrte;     /* the offset (in chars) to the mark location */
} Position;
```

- TODO: Define the datatypes to be used in the code.

PART I - READER STRUCTURE EXPLANATION

Where:

- ***content*** is the pointer that indicates the beginning of useful information (loaded from a source file).
- ***size*** 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 size. It is whatever value you have used in the call to ***malloc()/realloc()*** that allocates the storage pointed to by ***content***.

- ***increment*** is an increment factor. It is used in the calculations of a new **reader size** when the **reader** needs to grow. The **reader** needs to grow when it is full but still another character needs to be added to the **reader**. The **reader** is full when ***wrte*** measured in bytes is equal to ***size*** and thus all the allocated memory has been used. The ***increment*** is only used when the **reader** operates in one of the "self-incrementing" modes and it must respect the integer definition of your language.
  - In "**additive** self-incrementing" mode the integer data increment is used in ADDITION to the current size.
  - In "**multiplicative** self-incrementing" the integer data increment is used to MULTIPLY the current value.
  - In "**fixed**" mode, once defined the size, the reader size cannot increase.
- ***mode*** is an operational mode indicator. It can be set to three different values that must be defined in your **reader**: given by the **enum** `READER_MODE`).
- ***flags*** is a field containing different flags and indicators.
  - 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 **reader**, integer arithmetic sign overflow, and so on).
    - Multiple-bit indicators can indicate, for example, the mode of the **reader** (three different combinations – therefore 2-bits are needed). In this implementation the ***flags*** field has the following structure:
  - In the current implementation, we have:

Bit 7 (MSB)							Bit 0 (LSB)
-	-	-	-	**FUL**	**EMP**	**REL**	**END**

  - Note that the sequence of 0000.0000 (1 byte) correspond to the value **00** hexadecimal. In short:
    - It means that the `READER_DEFAULT_FLAG` (initially created) is 00.

Note 5: Understanding 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 **reader**, file **reader**, and database fields.

The flags are usually manipulated through different bitwise operations using a set of "masks." Alternative technique is to use bit-fields. They allow individual fields to be manipulated in the same way as structure members are manipulated.

In this implementation, you are to use bitwise operations (SET / RST / CHK) for setting, resetting and checking the content of the **reader**. Remember that since in C, it is not possible to access one single bit, you need to use masks.

  - The **bit 0** (***END***) indicates the End Of the Reader, having also default value as **0**, and when set to **1**, it indicates that the end of the reader content has been reached during the **reader** read operation (*bufGetChar()* function). If ***END*** is set to **1**, the function *bufGetChar()* should not be called before the *posRead* is reset by another operation.

        - When you have read the **reader**, once you achieve the end of the **reader**, the last bit (END) must be SET.
    - The **bit 1** is called ***REL*** **(relocation bit)** and signalizes that your **reader** content has been reallocated. It is by default **0**, and when set to **1**, it indicates that the location of the reader character array in memory has been changed due to memory reallocation.
        - This can happen when the **reader** 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 **reader**.
        - When the **reader** reallocation is called, if it is created a new memory position, the REL bit must be SET.
    - The **bit 2** (***EMP***) is used to identify that the **reader** is empty (ex: when it is created), and when set to **1**, it indicates that reader content is empty.
        - You need to adjust it appropriately when creating and updating the **reader**. Use also check bit comparison in the function bufIsEmpty().
    - The **bit 3** (***FUL***) is used to identify that **reader** is full (ex: when the position given by posWrte has achieved the MAX_SIZE), and when set to 1, it indicates that reader content is not possible to include chars.
        - Note that the size is increased by one of the modes: additive or multiplicative). You need to adjust it appropriately when creating and updating the **reader**. Use also check bit comparison in the function bufIsFull().
    - The remaining bits can be 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.

- ACTION: Define the values for bit-wise operations using the flags in the **reader** structure to accommodate a *NEW definition:*

```
/* Add your bit-masks constant definitions here */
/* TO_DO: BIT 3: FUL = Relocation */
/* TO_DO: BIT 2: EMP = Empty */
/* TO_DO: BIT 1: REL = Relocation */
/* TO_DO: BIT 0: END = End Of Reader */
```

- **Attention**: the operations over bits (SET, RST and CHK) will be explained during lectures and are also described in the **Lecture Notes**.

Note 6: About Masks and Operations
Basically, for each bit you can perform some operations. For example: SET: means that you "set" 1 to one specific bit, RST: you "reset" the specific bit to 0 and CHK for checking the value of one specific bit (that can be 0 or 1). These masks are also used with operations for getting the correct answer. Check the **Lecture Notes** (Annex 3).

❖ ***histogram*** is the last field that records the number of each kind of char found in the Reader.

- Note that there are `NCHAR` chars (using ASCII table, and the signed values for chars, it is possible to update 128 chars (from code 0 to 127).

PART II – POSITION STRUCTURE EXPLANATION

About the offsets, we have:

- ***wrte*** is the distance (measured in chars) from the beginning of the character array (*content*) to the location where the next character is to be added to the existing **reader** content.
  - *posWrte* must never be larger than ***size***, or else you are overrunning the **reader** in memory and your program may crash at run-time or destroy data.
- ***read*** is the distance (measured in chars) from the beginning of the character array (*content*) to the location of the character which will be returned if the function *getChar****()*** is called.
  - The value *posRead* must never be larger than *posWrte*, or else you are overrunning the **reader** in memory and your program may get wrong data or crash at run-time. If the value of *posRead* is equal to the value of *posWrte*, the **reader** has reached the **end** of its current content.
- ***mark*** is the distance (measured in chars) from the beginning of the character array (*content*) to the location of a specific position ("marked" to be used later).
  - A *mark* is a location in the **reader**, 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 the reader.

- TODO: Check all activities to be done in the code related to activities:
  - All constant definitions, data type and function declarations (prototypes) must be located in the **reader** header file (remember to rename according to your language name).
    - **TIP_01**: You need to read the following section (about functional specification) to adjust the values appropriately.
  - In the end of the header, you can see all function declarations that will be used in **reader**.

Reader test

Note 7: About Files in Brightspace
When you download the files, you will see that the compilation must be completely well succeed. So, when you update your files, you need to have a "perfect" compilation – no errors / warnings. One additional note: the code running is not COMPLETE (even in SOFIA): the purpose is that you can review the logic and update according to the correct definition of the **reader**.

2.1. GENERAL VIEW – INPUT FILES

To test your program, you are to use the test harness program ***MainReader.c*** and test it with input files.

- **TIP_02**: In your main **reader** file, you must have just ONE reference to your header:

```
#include "Reader.h"
```

- For this reason, you have some constants in your **reader** header file that are considered "language neutral". For instance: READER_ERROR.

❖ ***Important Note:*** Because each project is implementing a specific language, you need to **create input files** (using your EXTENSION language name) that can be read by your language. For instance, you will see inputs such as:

- An empty file (**INPUT0_Empty.boa**);
- A Hello World file (**INPUT1_Hello.boa**);
- A formula to calculate volume (**INPUT2_Volume.boa**);
- A factorial calculus (**INPUT3_Factorial.boa**);
- A basic file using different datatypes (**INPUT4_Datatypes.boa**);
- A generic file with several statements in Sofia (**INPUT5_General.boa**);
- In this assignment (A12), a big file for testing (**INPUT6_Big.boa**).
  - And a big file (that maybe do not need to be a "real" code file) aims to test limits of your **reader**, passing parameters such as mode, size and increment (check the **reader** properties). For instance, look the way to create long strings (*note that this is really an enormous string*).
    - Ex: the well-known randomized words "Loren Ipsum" (see: https://www.lipsum.com/)
    - You can test **reader** modes and parameters with a "Big file" – it means that your input must have about 32.000 chars (less than your "integer" definition).

❖ ***Important Note:*** However, remember that your reader must be able to work with **ANY text file**.

❖ For standard tests, inputs can be tested using:

- By default, the **fixed** mode ("**f**") mode, that means you cannot increase the **reader** size.
- But you need also to test using the modes "**a**" (**additive**) and "**m**" (**multiplicative**) modes.

2.2. ABOUT PROJECT - Makefile

This time, we suggest (recommendation / not requirement) that your project in the IDE can be done using **Makefiles**.

Note 8: About Makefiles
The use of **make** as a tool for link and compile C files in one single project is a classical way to develop solutions (before the appearance of builders such as Maven). Basically, you define standard configurations and the files to compose the project (in our case, .C and .H files). Remember that this file must be updated for each new assignment.

Example:

```
# CMakeList.txt : CMake project for CompilersMake, include source and
# define project specific logic here.
cmake_minimum_required (VERSION 3.8)
```

```
project ("Compilers")
# Add source to this project's executable.
add_executable (Compilers
       "Compilers.h" "Compilers.c"
       "Reader.h" "Reader.c" "MainReader.c"
)
# TODO: Add tests and install targets if needed.
```

- The corresponding output files (remember to redirect the outputs) must be also provided:
  - Remember that the way you can generate must be similar to this:

<PROGRAM> <OPTION> <INPUT_FILE> > <OUTPUT_FILE> 2> <ERROR_FILE>

  - **Example:**
    - Suppose that your project is "Compiler".
    - The value "R" is related to the option to execute **reader**, as it will be shown in the next section.

Compiler.exe R **hello.rio** > hello.out 2> hello.err

- **About outputs**:
  - ACTION: So, for each input, provide the default output and the default error file:
    - You will find similar outputs for standard output (with **".out"** extension)
    - And also, for standard error (with **".err"** extension).
- ***Important Note:*** All files are supposed to have only plain ASCII characters.

Note 9: About Submission
When you submit, your input (using your extension language name), and output and error files must be included. It is expected that all the errors files must be empty.

2.3. ABOUT EXECUTION - Example

Here is a brief description of the program that is provided for you on Brightspace (BS).

- You will find a unique main file (**Compilers.c**) with the corresponding header file (**Compilers.h**).
  - This file is already prepared to execute all activities in this course (**A12** - **reader**, **A22** - **scanner** and **A32** - **parser**). So, you need to change it to adapt the script to your language.
  - The **reader** execution is given by this command (suppose that "**Naja**" (Egyptian snake) is the name of your compiler, using "naj" extension). Commands can be like this:

Compiler.exe R hello.brz > hello.out 2> hello.err

    - When "R" (argv[1]) is passed, mainReader(), the function in the **MainReader.c** file is invoked. The other options will be available in future assignments ("S" = Scanner and "P" = Parser).

- The first redirection is to create the output (in our case "hello.out");
- And the 2nd redirection, passing the parameter "2>" aims to redirect the errors to one specific file ("hello.err").

❖ The main **reader** program (**Compilers.c**) takes to parameters from the command line and for **reader** execution, it is required:

- An input file name and a character (**f** – ***MODE_FIXED***, **a** – ***MODE_ADDIT***, or **m** – ***MODE_MULTI***) specifying the **reader** operational mode.
- For testing additional files, include the size and the increment.

❖ ***NOTE:*** Your program must not overflow any **reader** in any operational mode, no matter how long the input file is.

❖ **TIP_03**: **In this assignment, the provided main program will not test all your functions.** You are strongly encouraged to test all your **reader** functions with your own test files and modified main function.

2.4. ABOUT BATCH

❖ Besides the code, you need to submit a batch file with the complete execution of all scenarios (empty file, hello, basic file – with different datatypes – and big file).

❖ For this reason, we are including one specific example in Sofia Language[2]:

```
:: COMPILERS COURSE - SCRIPT ---------------------------------------------
:: SCRIPT A12 - CST8152 - Fall 2022

CLS
SET COMPILER=Boa.exe

SET FILE1=INPUT0_Empty
SET FILE2=INPUT1_Hello
SET FILE3=INPUT2_Volume
SET FILE4=INPUT3_Factorial
SET FILE5=INPUT4_Datatypes
SET FILE6=INPUT5_General
SET FILE7=INPUT6_Big

SET ASSIGNMENT=A12
SET EXT=boa
SET OUTPUT=out
SET ERROR=err

SET PARAM=R

:: ------------------------------------------------------------------------
:: Begin of Tests (A12 - F22) -----------------------------------------
:: ------------------------------------------------------------------------
```

[2] Remember to adjust at least the following elements in this script: your compiler name, the language definition (and extension), the list of files to be tested (maybe you do not have 7 files), but remember to teste at least: [a] an empty file, [b] a hello file and [c] a big file.

```
@echo off
ECHO " _____________________________________ "
ECHO "|                                     |"
ECHO "| ....... 'BOA' LANGUAGE ........     |"
ECHO "|      __    __    __    __           |"
ECHO "|     /  \  /  \  /  \  /  \          |"
ECHO "|  __/   __\/   __\/   __\/   __\__    |"
ECHO "| _/  /__/  /__/  /__/  /_____|   |"
ECHO "|  \_/ \    / \    / \    / \  \___  |"
ECHO "|       \_/    \_/    \_/    \___o_> |"
ECHO "|                                     |"
ECHO "| .. ALGONQUIN COLLEGE - 2022F .. |"
ECHO "|_____________________________________|"
ECHO "                                       "
ECHO "[READER SCRIPT ...................]"
ECHO "                                       "

ren *.exe %COMPILER%

:: BASIC TESTS  ------------------------------------------------------------
:: Basic Tests (A12 - F22) - - - - - - - - - - - - - - - - - - - - - - - - -

%COMPILER% %PARAM% %FILE1%.%EXTENSION% > %FILE1%-%ASSIGNMENT%.%OUTPUT% 2>
%FILE1%-%ASSIGNMENT%.%ERROR%
%COMPILER% %PARAM% %FILE2%.%EXTENSION% > %FILE2%-%ASSIGNMENT%.%OUTPUT% 2>
%FILE2%-%ASSIGNMENT%.%ERROR%
%COMPILER% %PARAM% %FILE3%.%EXTENSION% > %FILE3%-%ASSIGNMENT%.%OUTPUT% 2>
%FILE3%-%ASSIGNMENT%.%ERROR%
%COMPILER% %PARAM% %FILE4%.%EXTENSION% > %FILE4%-%ASSIGNMENT%.%OUTPUT% 2>
%FILE4%-%ASSIGNMENT%.%ERROR%
%COMPILER% %PARAM% %FILE5%.%EXTENSION% > %FILE5%-%ASSIGNMENT%.%OUTPUT% 2>
%FILE5%-%ASSIGNMENT%.%ERROR%
%COMPILER% %PARAM% %FILE6%.%EXTENSION% > %FILE6%-%ASSIGNMENT%.%OUTPUT% 2>
%FILE6%-%ASSIGNMENT%.%ERROR%

:: ADVANCED TESTS  ---------------------------------------------------------
:: Advanced Tests (A12 - S22) - - - - - - - - -- - - - - - - - - - - - - -

%COMPILER% %PARAM% %FILE7%.%EXT% f 100 10 > %FILE7%-%ASSIGNMENT%-f-100-
10.%OUTPUT% 2> %FILE7%-%ASSIGNMENT%-f-100-10.%ERROR%
%COMPILER% %PARAM% %FILE7%.%EXT% a 100 10 > %FILE7%-%ASSIGNMENT%-a-100-
10.%OUTPUT% 2> %FILE7%-%ASSIGNMENT%-a-100-10.%ERROR%
%COMPILER% %PARAM% %FILE7%.%EXT% m 100 10 > %FILE7%-%ASSIGNMENT%-m-100-
10.%OUTPUT% 2> %FILE7%-%ASSIGNMENT%-m-100-10.%ERROR%

:: SHOW OUTPUTS - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
DIR *.OUT
DIR *.ERR

:: ---------------------------------------------------------------------
:: End of Tests (A12 - F22) ------------------------------------------
:: ---------------------------------------------------------------------
```

- In the ASSIGNMENT SUBMISSION STANDARD (**CST8152_Compilers_S22-ASSAMG**), it is mentioned that you must include:

- The code with the adaptation for your language.
- Inputs, standard outputs and error output files.
- **Test Plan**: The way you can show / describe how to execute your program (ex: script / batch or simply command line arguments).

- Finally, remember to avoid problems with warnings, late submission or plagiarism.

Outputs: Are typically like this…

```
>BoaCompiler.exe R README.txt
 ______________________________________
|                                      |
| ....... 'BOA' LANGUAGE ........      |
|      __    __    __    __           |
|     /  \  /  \  /  \  /  \          |
| __/  __\/  __\/  __\/  __\__        |
| _/  /__/  /__/  /__/  /_____|       |
|  \_/ \   / \   / \   / \  \___      |
|       \_/   \_/   \_/   \___o_>      |
|                                      |
| .. ALGONQUIN COLLEGE - 2022F ..      |
|______________________________________|

[Option 'R': Starting READER .....]

Reading file README.txt ....Please wait

Printing buffer parameters:

The capacity of the buffer is:  250
The current size of the buffer is:  106
The operational mode of the buffer is: f
The increment factor of the buffer is:  10
The first symbol in the buffer is:  #
The value of the flags field is: 00
Number of different chars read: 53

Printing buffer contents:

# 'BOA' Example
  ABCDEFGHIJKLMNOPQRSTUVWXYZ #
main& {
        data {
        }
        code {
                print&('Hello world!');
        }
}
```

Appendix – Explaining Functionalities

- ***IMPORTANT NOTE:*** Since the code has been provided for SOFIA, just adjust the **Reader** functionalities to execute in your language (see the "TODO" comments in code).

1.4. READER FUNCTIONALITIES (Reader.c)

You are to implement the following set of **reader** utility functions (operations). All function definitions must be stored in a file named by ***Reader.c***. Later they will be used by all other parts of the compiler when a temporary storage space is needed.

The **first implementation step** is to adjust the inclusion of header file. 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 10: 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. For instance, evaluate all important parameters that you are receiving in one specific function, before performing one action.

1.4.1. General Operations

Now, we will explain each function in **reader** (the code has already been started for you):

- `readerCreate` ***(size, increment, mode);***
  - This function creates a new **reader** in memory (on the program heap), trying to allocate memory for one **reader** structure using ***calloc()***;
  - It tries to allocates memory for one dynamic character (***content***) calling ***malloc()*** with the given initial capacity ***(size)***;
  - The **reader** is supposed to use the parameters passed to initialize the fields size, increment and mode, but some adjustments must be done:
    - If there is no value for ***size***, default values (for size and increment) are used.
    - If there is no value for ***increment***, the ***mode*** must be fixed.
    - If ***mode*** is different from valid options (Fixed, Additive or Multiplicative), no **reader** must be created.
    - The ***flags*** must be initialized by default values defined in the header.
  - ***Return value***: Reader pointer.

- ***Flag***: Set values to default.
- ***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.

❖ `readerAddChar` ***(pReader, ch)***

- This is the most important function in the code. This function is responsible to include a char in the **reader**. Because of the limit (given by ***size***), several actions should be done before simply include it in the end of the **reader**.
  - Because there is a possibility that the **reader** can be already full, while the maximum size is not achieved (for instance, `READER_MAX_SIZE`), if the mode is not fixed, it is required to reallocate additional space (using C function to ***realloc***).
- If the **reader** can include a char (it is not full), just include it in the current position given by ***wrte*** offset and increment this position.
- Otherwise, if the reader is full, some actions must be done:
  - Start using a bitwise operation the function resets the ***RLB*** bit on ***flags***.
  - If mode is MODE_FIXED, it is not possible to allocate additional space and function ends.
  - If the mode is MODE_ADDIT, it is required to increase the size by using a linear formula:

    ```
    newSize = pReader->size + pReader->increment;
    ```

  - If mode is multiplicative (MODE_MULTI) the formula is given by:

    ```
    newSize = pReader->size * pReader->increment;
    ```

  - In both cases, it is required to check if the result is valid (not negative and lower than `READER_MAX_SIZE`) and in case of error, no inclusion can be done.
  - Otherwise, if it is detected that the memory address has been changed, the **REL** bit must be set (use comparison between addresses to check this).
  - If everything is ok, you can use ***realloc***() to increase the size of the ***content***.
    - However, remember to use *defensive programming*: call realloc using a temporary variable and avoid to destroy the original **reader**.
  - Finally, add the character ***ch*** to the character array of the given ***content*** pointed by ***pReader***.
- ***Return value***: Reader pointer.
- ***Flag***: You need to manipulate the flags carefully:
  - In order to check if it is possible to include a char (without reallocation), check the FUL bit.
  - Due the risk that it is possible to do memory reallocation, start setting the RLB bit.
  - When adding the char, if the newSize is not valid (remember what does it mean), set the FUL bit.
  - When doing the reallocation (remember each mode), if the memory position has been changed (think about how to discover this by comparison), set RLB bit.

- ***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.
  - **TIP_02**: The function ***must not destroy*** the **reader** or the contents of the **reader** even when an error occurs – it must simply return NULL leaving the existing **reader** content intact.

Note 11: Implementing Modes
Remember, the **reader** is a **space** that can be updated to include chars. However, you need to respect the modes ('f', 'a' or 'm'): once achieved the capacity (size), if the mode is fixed, it is no more possible to increase it. In the case of additive or multiplicative, you can do, using the formulas – in the first case, you add the value of increment and in the second one, you multiply the current size by increment. However, remember to check if you did not get an overflow (negative values may appear). Otherwise, you can continue including chars.

❖ `readerClear` ***(pReader)***

- The function retains the memory space currently allocated to the **reader**.
- It reinitializes all appropriate data members of the given **reader** structure so that the **reader** will appear as just created.
- In short, offsets and flags must be reset.
- ***Flag***: When you are cleaning the **reader**, adjust all bits from flag (similar during creation).
- ***Return value***: Boolean value defined by your language.

❖ `readerFree` ***(pReader)***

- The function de-allocates (frees) the memory occupied by the character **reader** and the **reader** structure.
- ***Return value***: Boolean value defined by your language.

Note 12: Correct destruction
Check if you are in fact "freeing" memory once you do not need to use the **reader** more.

❖ `readerIsFull` ***(pReader)***

- The function checks if the character **reader** is full.
  - Basically, it is required to evaluate the bit-wise operation for checking the value of the bit flag ***FUL*** from flags.
- ***Flag***: You can use the CHK mask with and evaluate the FUL bit.
- ***Return value***: Boolean value defined by your language.

- `readerIsEmpty` ***(pReader)***
  - This function checks if the **reader** is empty.
    - Basically, it is required to evaluate the bit-wise operation for checking the value of the bit flag ***EMP*** from flags.
  - ***Flag***: You can use the CHK mask with and evaluate the EMP bit.
  - ***Return value***: Boolean value defined by your language.

- `readerSetMark` ***(pReader, mark)***
  - The function sets a new value to ***posMark*** offset.
    - **Note:** Remember that we are talking about "valid" positions in the **reader**, so the value must be between 0 and the posWrte**.**
  - ***Return value***: Boolean value defined by your language.

Note 13: About Mark position
The posMark value will be very important in the Scanner and Parser: you need to understand that when you "mark" one position, you can use this value to return later, once some conditions are achieved – it is a kind of "memory" that you need to maintain (ex: while recognizing tokens).

- `readerPrint` ***(pReader)***
  - This function is intended to print the content of **reader** (in the ***content*** field).
  - Using the ***printf()*** library function the function prints character by character the contents of the character **reader** to the standard output (***stdout***).
  - You must use a loop that invokes `readerGetChar` ***()*** and **checking the flags** (***END*** - End Of the Reader) can print the content.
  - ***Flag***: During the reading, you can use END bit for checking if you had finished the process.
  - ***Return value***: The number of characters printed.

- `readerLoad` ***(pReader, fi)***
  - The function loads (reads) an input file specified by ***fi*** and put its content into a **reader** (the field content).
    - **Note:** The file is supposed to be a plain file (for instance, a source code).
  - **TIP_03**: The function must use the standard function ***fgetc(fi)*** to read one character at a time and the function `readerAddChar` ***()*** to add the character to the **reader**.
    - 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 **reader**.
    - The standard macro ***feof(fi)*** can be used to detect end-of-file on the input file.

- ***Return value***: The function returns the number of characters read from the file stream.
- ***Error Condition:*** If the current character cannot be added to the **reader** (`readerAddChar` ***()*** returns NULL. It is also necessary to use the ANSI C function ***ungetc()*** when if the function gets an error.
  - ***Note:*** This "***ungetc***()" operation is required because this latest char could not be included into the **reader**.

Note 14: About Load
This function is especially important to initialize the content of the **reader**: you need to be able to manipulate all files (the only requirement is that you can read only ASCII chars) and then, you can include them the content in the **reader**. If you are in a fixed mode or, eventually, the size of file is too big, the content can be truncated. Remember that you do it char by char, invoking the `readerAddChar` ().

❖ `readerRecover` ***(pReader)***

- The function sets both ***read*** and ***mark*** offset to 0, so that the **reader** can be reread again.
- ***Return value***: Boolean value defined by your language.

❖ `readerRetract` ***(pReader)***

- The function simulates the operation to "unread" one char from the **reader**.
  - It is a logical function that decrements ***read*** offset by **1**.
  - It means that it is required to be sure that ***read*** is positive.
- ***Return value***: Boolean value defined by your language.

❖ `readerRestore` ***(pReader)***

- The function sets ***read*** offset to the value of the current ***mark*** offset.
  - It will be used when it is necessary to "undo" several reading operations (and, for this reason, ***mark*** offset is used).
- ***Return value***: Boolean value defined by your language.

1.4.2. Getters

❖ `readerGetChar` ***(pReader)***

- Another important function. It is used to read the **reader**. The function performs the following steps:
  - To get the current char during the read process, the ***readPos*** offset is used (and incremented).
  - If ***read*** and ***wrte*** positions are equal, using a bitwise operation it sets the ***flags*** field ***END*** bit flag to **1** and returns number the end of string (**'\0'**); otherwise, using a bitwise operation resets ***END*** to **0.**

- ***Flag***: You are supposed to set END when you have reached the end of the **reader**.
- ***Return value***: The character located at ***read*** position.

❖ `readerGetContent` ***(pReader, pos)***

- The function returns a pointer to the location of the character **reader** indicated by ***pos*** that is the distance (measured in chars) from the beginning of the character array (***content***).
  - Because you can define the position, it is required to test it: the ***pos*** value must be between 0 and ***wrte*** position.
- ***Return value***: A pointer to char (string in C) that starts with the position ***pos*** after the ***content*** reference in **reader**.

❖ `readerGetPosRead` ***(pReader)***

- The function returns the ***read*** value to the calling function.
- ***Return value***: Integer that represents the ***read*** position.

❖ `readerGetPosWrte` ***(pReader)***

- The function returns the current number of chars included in the **reader** (given by the ***wrte*** offset)
- ***Return value***: Integer value defined by your language.

❖ `readerGetPosMark` ***(pReader)***

- The function adjusts the ***mark*** position.
- ***Return value***: Boolean value for this operation.

❖ `readerGetMode` ***(pReader)***

- The function returns the current ***mode*** (fixed / additive / multiplicative).
- ***Return value***: Value defined by your language.

❖ `readerGetSize` ***(pReader)***

- The function returns the value of ***size*** of the **reader** to the calling function.
- ***Return value***: Integer that represents the ***size*** of the **reader**.

❖ `readerGetInc` ***(pReader)***

- The function returns the value of ***increment*** to the calling function.
- ***Return value***: Integer that represents the ***increment*** value.

- `readerGetFlags` ***(pReader)***
  - The function returns the ***flags*** field from **reader**.
    - Note that the entire flag must be returned (not only one specific bit).
  - ***Return value***: The value of ***flags*** in the appropriate datatype defined by your language.

- `readerShowStat` ***(pReader)***
  - The function returns the number of different chars found in the **reader**.
    - Note that the field ***histogram*** is a vector of integers that indicates for each position (from 0 to `NCHAR`) the amount of each char.
  - ***Return value***: The number of different chars found in the **reader**.

Evaluation

- Please read the Assignment **Submission Standard and Marking Guide** (at "Assignments > Assignment Guide" section).
  - 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.
  - You are receiving a code using **BOA language** that can be used as a model
- **About Plagiarism**: Your code must observe the configuration required. Similarly, we need to observe the policy against ethic conduct, avoiding problems with the 3-strike policy...

Submission Details

- **OPTION 1: Digital Submission: Compress** into a **zip** file with the complete code (.C, .H, Makefiles and **scripts**) that you created specifying the language and include as attachment in the *BrightSpace*.
  - ***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: **A11_Sousa123.zip**.
    - If you are working in teams, please, include also your partner. For instance, something like: **A11_Sousa123_Melo456.zip**.
- **OPTION 2: GitHub Submission:** In the *BrightSpace*, include just the repository links (giving access to the professor if it is not public).

Note 5: The GitHub utilization.
*This term, we are introducing the use of GitHub as a bonus activity (see the additional marks in the Rubric). So, you can simply submit a minimal instruction (ex: README.TXT) describing the access to your repository, as well as the **hash code** for your version.*

- ***IMPORTANT NOTE:*** **Assignments will not be marked if there are not source files in the digital submission / GitHub repository.** 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.

Assignment evaluation

Marking Rubric

Maximum Deduction (%)	Deduction Event
CRICTICAL	**Severe Errors**
5 pt (100%)	Late submission (after 1 week due date)
5 pt (100%)	Plagiarism detection (code from previous versions)
2.5 pt (50%)	Does not compile in the environment (considering **Visual Studio 2022**)
2.5 pt (50%)	Compile but crashes after launch
PATTERN	**Non-compliance**
2 pt	Language adaptation (missing elements in the Reader)
1 pt	Missing files (input)
1 pt	Missing script (test plan)
BASIC	**Execution**
0.5 pt each	Problems with small files (empty, hello, basic)
0.5 pt each	Big file errors (problems with modes – 'f', 'a', 'm')
ADDITIONAL	**Small problems**
0.1pt each	Warnings[3]
1.0	Missing defensive programming
0.5	Other minor errors
1 pt	GitHub
1 pt	Clean and elegant code, enhancements, discretionary points.
Final Mark	**Formula: 5***((100- ∑ penalties + bonus)/100), max score 7%**.**

File update: Sep 18th 2022.

Good luck with A12!

[3] **HELP for students**: Eventual problems during execution – maybe caused by different IDEs, operational systems, or C compiler version – can also be compared with additional resources (ex: images) provided by students that can demonstrate that the original code was able to run without problems.