A Non-Overflowing Dynamic Buffer Utility

Description

This buffer program applies programming techniques, data types and structures, memory management, and simple file input/output. It also applies “excessively defensive programming”, and shows the type of internal data structures used by a simple compiler.

Buffers are often used when developing compilers because of their efficiency. This is 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.

The following structure declaration is 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;
```

Definitions:

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 the value 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 is 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 is 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) is never be larger than ***capacity***, or else it will be are overrunning the buffer in memory and the 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.

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 is 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.

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 is set to 1 and stay 1 all the time in this implementation														**eob flag**	**r**_flag

Bit 0 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 LSB bit (bit 1) 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()*** is not called before the ***getc_offset*** is reset by another operation.

The rest of the bits are reserved for further use and is set by default to **1.** When setting or reseting ***eob*** or ***r_flag*** they are not changed by the bitwise operation manipulating bit 0 and 1.

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* is 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*. If the *init_capacity* is **0,** the function tries to create a character buffer with default size **200** characters. If the *init_capacity* is **0,** the function ignores the current value of the parameter *inc_factor* and sets the buffer structure ***inc_factor*** to **15** in mode ***a*** and ***m*** or to **0** in mode ***f***. The pointer returned by ***malloc()*** is assigned to ***cb_head***;
- sets the Buffer structure operational mode indicator ***mode*** and the ***inc_factor***. If the *o_mode* is the symbol ***f*****,** the ***mode*** and the buffer ***inc_factor*** are set to number **0**. If the *inc_factor* parameter is **0** and *init_capacity* is not **0** (see above)**,** 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.

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.

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 returns **–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 does 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 returns **–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 returns **–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 returns **–1**.

short b_mark (pBuffer const pBD, short mark)
The function sets ***markc_offset*** to ***mark***. The parameter ***mark*** is 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 returns **-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 notifies 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 returns 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 uses 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 is not be added to the content of the buffer.
Only the standard macro ***feof(fi)*** is 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 returns **–1**. If the loading operation is successful, the function returns 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 returns **–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 value of the ***flags*** field determined only by the ***eob*** bit.
A bitwise operation is used to return the value of the ***flags*** field. If a run-time error is possible, it returns **–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 returns **0**. If the buffer is not empty, the function prints the content calling ***b_getc()*** in a loop and using ***b_eob()*** 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). 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 returns NULL if for some reason it cannot to perform the required operation. It sets the ***r_flag*** bit appropriately***.***

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

short b_retract (Buffer * const pBD)
The function decrements ***getc_offset*** by **1**. If a run-time error is possible, it returns **–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 returns **–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 returns **–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 returns **–1**; otherwise it returns 0;

char * b_location(Buffer * const pBD)
The function returns a pointer to a location of the character buffer indicated by the current ***markc_offset***. If a run-time error is possible, it returns **NULL**.