COMP2401 - Assignment #2

(Due: **Wednesday, February 7th, 2024 @ 11pm**)

In this assignment, you will work with arrays of strings and pack data into bytes based on bits. Then you will unpack them.

Data compression is an important topic in computer science because as time goes on, more and more data becomes available and there is always a need to reduce the amount of storage required to save the data. In this assignment, you will implement a VERY SIMPLE strategy for compressing some basic text from 8 bits to 6 bits.

As you know, each byte (i.e., **unsigned char**) in C is represented by an 8-bit binary value associated with it. So, when we read in characters from the user, they are represented using ASCII code values. You will create a quick encoder that will reduce text by trimming 8-bit bytes down to a 6-bit representation. Since **6** bits only allows **64** values … we will encode only the letters, digits and a few other characters and all other characters will be represented by a single common character. Here is how we will map the characters to a 6-bit value:

CHARACTER	8-BIT ASCII VALUE	6-BIT VALUE
NULL	0	0
A-I	65-73	1-9
a-i	97-105	1-9
ENTER (i.e., CR)	10	10
J-Z	74-90	11-27
j-z	106-122	11-27
[	91	28
]	93	29
@	64	30
various symbols	32-63	32-63
all other characters	64-255	31

(1) Download the six files provided with this assignment (i.e., **convertTo6bit.c**, **usefulTools.c**, **usefulToos.h**, **compress.c**, **expand.c** and **convertToASCII.c**)

The program called **convertTo6bit.c** has been started for you. It reads the command-line arguments to the program (more on this later in the course) to decide if the debugging information should be shown. It then asks the user for a string of at most 255 characters. **You will insert code to convert the 8-bit ASCII characters from that string into a 6-bit character string by using the mapping shown in the above table**. The program then displays debugging information (if enabled) and finally sends the 6-bit encoded sentence to the console output. The code will make use of some functions provided in the **usefulTools.c** file … so you compile the program as follows:

```
gcc -o convertTo6bit convertTo6bit.c usefulTools.c -lm
```

Then run the program as follows:

```
./convertTo6bit -d
```

The program should display the 6-bit codes for each entered character and also display the string result (which will contain some characters that cannot be displayed). Here is some sample output. Make sure that these cases all work fine (also make sure that a **0** is always at the end of the output string):

User Input	Displayed Codes	Visible Part of Converted String
`Aa 19 @.`	`01 01 32 49 57 32 30 46 00`	` 19 .`
`A0A0A0A0`	`01 48 01 48 01 48 01 48 00`	`0000`
`I Love This Assignment!`	`09 32 13 16 23 05 32 21 08 09 20 32 01 20 20 09 07 15 14 05 15 21 33 00`	`!`
`[abyz] ABYZ.@0189%~+=;`	`28 01 02 26 27 29 32 01 02 26 27 46 30 48 49 56 57 37 31 43 61 59 00`	`��189%+=;`
`.`	`46 00`	`.`

Run the program without the **`-d`** option (i.e., **`./convertTo6bit`**). You should not see the byte codes but just the final converted string.

(2) The program called **compress.c** has also been started for you. After reading the command-line arguments it then reads in a string of converted characters from the **convertTo6bit** program. **You will insert code to pack the 6-bit character codes from the input string into 8-bit bytes (stored as an output string) so that the storage space required is reduced**. The program then displays debugging information (if enabled) and finally sends the compressed bytes to the console output as a string.

Consider the input string `"Test."`. The conversion of his string results in the 6-bit codes shown here on the right → along with their binary representation as stored in each of the 5 bytes.

6-bit code stored in a byte

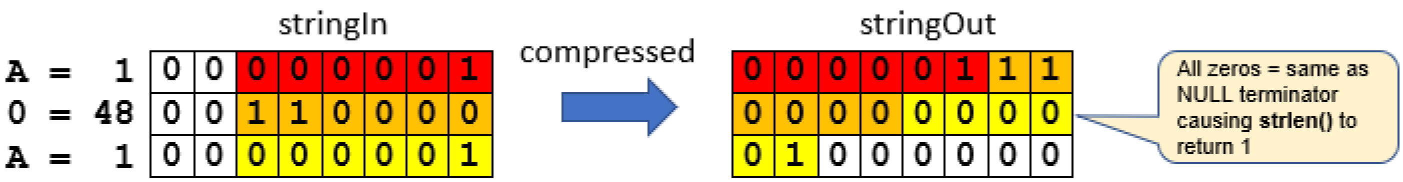
T = 21	0	0	0	1	0	1	0	1
e = 05	0	0	0	0	0	1	0	1
s = 20	0	0	0	1	0	1	0	0
t = 21	0	0	0	1	0	1	0	1
. = 46	0	0	1	0	1	1	1	0

To obtain the compressed output string bytes, you will need to take the 6 bits from each of the 5 input string bytes and pack them into the 8-bit bytes of the output string. To do this, just copy the 6 bits from the first character into the first byte of the output string. There will be two bits remaining in the output string. Copy the first two bits from the next input string character into the output string to complete the first output string byte. Then begin the second output string byte by copying in the remaining 4 bits from the second input string byte. Keep copying bits in this manner … filling up each output string byte and then moving on to the next one. As a result, the output string will have just 4 bytes with all the bits used except for the last two bits, which should be set to 0. Here is the result on the right →
This is an 80% compression ratio (i.e., the output string is 80% of the size of the original string).

8-bit output string bytes

0	1	0	1	0	1	0	0
0	1	0	1	0	1	0	1
0	0	0	1	0	1	0	1
1	0	1	1	1	0	0	0

You will need to keep track of how many bytes you are writing to the output string. You will not be able to use **strlen()** to determine the size of the output string because multiple bits could pack into a byte as all **0**'s, which will appear as a NULL-terminator to **strlen()** … causing it to return a smaller value that the string's size. Here is an example that shows how this can happen when a simple string such as `"A0A"` is entered:



You will compile the program as follows:

```
gcc -o compress compress.c usefulTools.c -lm
```

Then run the two programs together as follows: `./convertTo6bit | ./compress -d`

This will run the **convertTo6bit** program first and the **|** character tells the shell window to use the output from that program as input to the **compress** program. That will allow us to type in a string of characters and see the resulting bits that get packed. Here is some sample output. Make sure that these cases all work fine:

User Input	Displayed Output		Visible Part of Converted String
`Aa 19 @.`	Compression ratio = 75.0% Before compression: 000001 000001 100000 110001 111001 100000 011110 101110	 After compression: 00000100 00011000 00110001 11100110 00000111 10101110	1��
`A0A0A0A0`	Compression ratio = 75.0% Before compression: 000001 110000 000001 110000 000001 110000 000001 110000	 After compression: 00000111 00000000 01110000 00000111 00000000 01110000	pp
`I Love This Assignment!`	Compression ratio = 78.3% Before compression: 001001 100000 001101 010000 010111 000101 100000 010101 001000 001001 010100 100000 000001 010100 010100 001001 000111 001111 001110 000101 001111 010101 100001	 After compression: 00100110 00000011 01010000 01011100 01011000 00010101 00100000 10010101 00100000 00000101 01000101 00001001 00011100 11110011 10000101 00111101 01011000 01000000	&P\X � E �=X@

[abyz] ABYZ.@0189%~+=;	**Compression ratio = 77.3%** **Before compression:** 011100 000001 000010 011010 011011 011101 100000 000001 000010 011010 011011 101110 011110 110000 110001 111000 111001 100101 011111 101011 111101 111011	 **After compression:** 01110000 00010000 10011010 01101101 11011000 00000001 00001001 10100110 11101110 01111011 00001100 01111000 11100110 01010111 11101011 11110111 10110000	p�m� ��{ x�W���
.	**Compression ratio = 100.0%** **Before compression:** 101110	 **After compression:** 10111000	�

(3) The program called **expand.c** has also been started for you. It reads in an 8-bit compressed string from the **compress** program. You will insert code to expand the compressed bytes back into the 6-bit character codes that were originally output from the **convertTo6bit** program (i.e., you will do the reverse of the **compress** program). The program then displays debugging information (if enabled) and finally sends the expanded bytes to the console output as a string. This time, you will be able to se the **strlen()** function to determine the number of bytes in the output string. You will compile the program as follows:

```
gcc -o expand expand.c usefulTools.c -lm
```

Then run the three programs together as follows:

```
./convertTo6bit | ./compress | ./expand -d
```

Here is some sample output. Make sure that these cases all work fine:

User Input	Displayed Output		Visible Part of Converted String
Aa 19 @.	**Expansion ratio = 133.3%** **Before expansion:** 00000100 00011000 00110001 11100110 00000111 10101110	 **After expansion:** 000001 000001 100000 110001 111001 100000 011110 101110	19 .
A0A0A0A0	**Expansion ratio = 133.3%** **Before expansion:** 00000111 00000000 01110000 00000111 00000000	 **After expansion:** 000001 110000 000001 110000 000001	0000

	01110000	110000 000001 110000	
I Love This Assignment!	**Expansion ratio = 127.8%** **Before expansion:** 00100110 00000011 01010000 01011100 01011000 00010101 00100000 10010101 00100000 00000101 01000101 00001001 00011100 11110011 10000101 00111101 01011000 01000000	 **After expansion:** 001001 100000 001101 010000 010111 000101 100000 010101 001000 001001 010100 100000 000001 010100 010100 001001 000111 001111 001110 000101 001111 010101 100001	!
[abyz] ABYZ.@0189%~+=;	**Expansion ratio = 129.4%** **Before expansion:** 01110000 00010000 10011010 01101101 11011000 00000001 00001001 10100110 11101110 01111011 00001100 01111000 11100110 01010111 11101011 11110111 10110000	 **After expansion:** 011100 000001 000010 011010 011011 011101 100000 000001 000010 011010 011011 101110 011110 110000 110001 111000 111001 100101 011111 101011 111101 111011	��189%+=;
.	**Expansion ratio = 100.0%** **Before expansion:** 10111000	 **After expansion:** 101110	.

(4) Finally, the program called **convertToASCII.c** has also been started for you. It reads in the 6-bit code expanded string from the **expand** program. You will insert code to convert these 6-bit character codes back into ASCII codes (i.e., reverse of **convertTo6bit** program). The output string size will have the same as the input string. All 6-bit values of 31 represent all ASCII characters that were not able to be represented in 6-bits and when converted back to ASCII, they should be converted to the '_' (i.e., underscore) character. You will compile the program as follows:

```
gcc -o convertToASCII convertToASCII.c usefulTools.c -lm
```

Then run the four programs together as follows:

```
./convertTo6bit | ./compress | ./expand | ./convertToASCII -d
```

Here is some sample output. Make sure that these cases all work fine (also make sure that a **0** is always at the end of the output string):

User Input	Displayed Codes	Visible Part of Converted String
`Aa 19 @.`	65 65 32 49 57 32 64 46 00	`AA 19 @.`
`A0A0A0A0`	65 48 65 48 65 48 65 48 00	`A0A0A0A0`
`I Love This Assignment!`	73 32 76 79 86 69 32 84 72 73 83 32 65 83 83 73 71 78 77 69 78 84 33 00	`I LOVE THIS ASSIGNMENT!`
`[abyz] ABYZ.@0189%~+=;`	91 65 66 89 90 93 32 65 66 89 90 46 64 48 49 56 57 37 95 43 61 59 00	`[ABYZ] ABYZ.@0189%_+=;`
`.`	46 00	`.`

Run the program without the **-d** option. You should not see the byte codes but just the final converted string.

Now that all the programs are working … here are 5 user strings that you can try to run through all 4 programs. You can cut/paste each one:

```
Q: Did you hear about the racing snail who got rid of his shell? A: He thought it
would make him faster, but it just made him sluggish.

Q: How does a mathematician induce good behavior in her children? A: `I've told you
n times, I've told you n+1 times...'

Have you heard of that new band "1023 Megabytes"? They're pretty good, but they
don't have a gig just yet.

There are only 10 kinds of people in this world: those who know binary and those
who don't.

"Knock, knock."  "Who's there?"   very long pause….  "Java."
```

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit all of your **.c source** files and all other files needed for testing/running your programs. DO NOT TAR your files. Make sure that your name and student number is in each source file at the top as a comment.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).