CSC236 L2: Bytes

This lab is designed for individual work.

This lab is designed for individual work, but you may consult with others as long as you report having done so.

Note that this assignment was created by Dr. Jan Pearce of Berea College.

Bits, Bytes, and Binary

This lab is all about bits, bytes, and binary. It is also about encoding information, using encoding for the detection of errors, and using encoding for cool effects.

Encoding is simply the process of converting data from one form to another. All written language is an encoding of sound information into a written format. English speakers can encode sounds as combinations of letters (a, b, …, z), and those who speak Mandarin typically encode groups of sounds as characters (i.e. 中 国). In contrast to human speech, all information stored in a computer is encoded in **binary**, which has an alphabet of two characters, each called a bit for **BI**nary Digi**T**.

Why should encoding make any difference? Humans think using abstract symbols, so we think that the letter J differs from the binary number 01001010 or from the decimal number 74. Yet, these concepts are all encoded the same way in C++ with differing data types. This is why data types matter so much in C++!

If we delve further into binary representation, a computer uses that because binary is a base-2 number system in which values are represented by different combinations of bits represented by any two characters. The use of 0 and 1 is common for human-readable symbols, OFF and ON is used when thinking of signals in a machine, and false and true is used when thinking of logical operations. In a computer, it is implemented via low and high voltage.

For counting, most modern human cultures use the decimal, or base-10, number system, so binary may seem weird to us, but it is not new to humans. Australia's aboriginal peoples counted by 2, not by 10. The binary system can be used for more than just processing numbers. Again, this idea is not new. Numerous tribes of the African bush sent complex messages using drum signals at high and low pitches. Morse code uses two digits (long and short) to encode the entire English alphabet. Some native American tribes used binary in smoke signals, each smoke puff was punctuated by no smoke, which is also binary.

In the computer, it is the primary "alphabet" for the transmission of all digital information (text, music, photos, videos).

Using Bytes and Characters and ASCII

A **byte** is a set of bits, most commonly eight bits. Historically, the byte was the number of bits used to encode a single character of text in a computer in English, and for this reason it is the smallest

addressable unit of memory in many computers. Think about why memory is measured in X-bytes, such as Giga Bytes or Mega Peta Bytes. For this assignment, consider a byte to be 8-bits in which the final bit might (or might not be “special”.) We will use a byte to represent characters.

The following is a table of **ASCII values**, where ASCII is abbreviated from American Standard Code for Information Interchange, which, as we have seen before, is a character encoding standard for electronic communication. You read the following table by finding the character that you want to encode and then concatenating the leftmost row identifier with the topmost column identifier. So, for example, the character 'J' is encoded as 1001010 (100+1010) in ASCII, and the character 'S' is encoded as 1010011 (101+0011).

	---0000	---0001	---0010	---0011	---0100	---0101	---0110	---0111	---1000	---1001	---1010	---1011	---1100	---1101	---1110	---1111
000----	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
001----	DLE	DCL	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
010----	space	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
011----	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
100----	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
101----	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
110----	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
111----	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Note that this requires only 7 bits, so the final bit in a byte can be repurposed! For what, you ask?

Error Detection using a Parity Bit

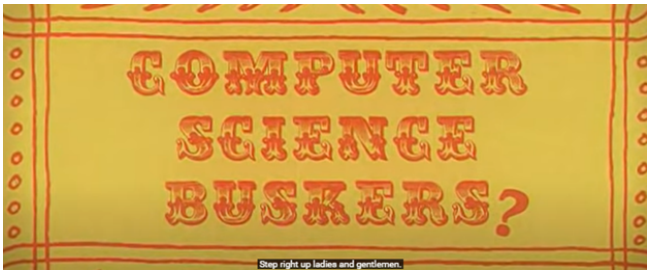
When data is transmitted, it is subject to noise which may introduce errors into the data. Error detection techniques such as the use of a **parity bit** facilitates the detection of such errors with the addition of a single bit. There are two types of parity (even and odd).

We will choose to explain the concept using even parity, but the ideas are similar for odd parity.

An **even parity bit** uses the last bit of a sequence of binary code as a simple form of error detection. The actual data is contained in all but the last digit. If the total number of true (or 1) bits in the actual data bits is odd, the parity bit value is set to true (or 1). Note that setting this bit makes the total count of trues or 1's in the sequence an even number. If the total number of true (or 1) in the actual data is already even, the parity bit's value is set to off (or 0), again keeping the number of actual true (or 1) values even.

Consider the following video that explains this notion in a fun way. Think of the green sides of the cards as 1 and the white sides as 0.

[Computer Science Buskers? (Error Detection)]



Using Parity for Error Detection with 7-Bit ASCII Character Codes

In modem-type communications, a parity bit is added to one end of the 7-bit ASCII character data described above, making it 8 bits, or 1 byte, long. For example, recall from before how the character 'J' is 1001010 (100+1010) in ASCII binary and only 7 bits of a byte stores the actual value. This feature exists for all characters, such as 'S', which is 1010011 (101+0011). Hence, the last bit in the byte is available for something else like parity error detection.

When we use even parity error detection, we add an even parity bit as this last bit. Because the letter 'J' has only 3 ones in its ASCII representation (1 - - 1 - 1 -) so using the logic described above, we would set the parity bit to 1, so that letter is encoded as 1001010**1** (100 and 1010 and 1), thereby ensuring that an even number of 1's are in the binary representation.

When we consider the character 'S' with even parity, the number of 1's in the ASCII representation already has an even number of 1's, so the parity bit is set to zero and it is represented as 1010011**0** (101 and 0011 and 0).

If you are still confused, optionally, you could watch the following Error Detecting Code Parity Explained | Odd Parity and Even Parity.

Masking of Bits

Masking is the process of keeping or removing information for a given purpose. During COVID-times, we have certainly become familiar with masks. Pre-COVID, there was already a long history of using masks for fun. Masks were worn at masquerade balls in which an important part of the fun was to hide one's identity behind an elaborate costume and a facial mask, and they are still used in that same way today at Mardi Gras festivals.

The technical term masking is based on the idea that a mask hides all or part of a person's face. With digital information, **masking** is the act of applying a mask byte to your original byte.

Application of Bit Masking

We are all using bit masking in Zoom when we add a background to make us appear to be somewhere else, but it has been used in photography and gaming for a long time.

An application of bit masking is used in computer graphics; if an image is intended to be placed over a background, the transparent areas can be created by applying a binary mask. This way, for each intended image there are actually two bit sets: the actual image, in which the unused areas

are given a pixel value with all bits set to 0s, and an additional mask, in which the correspondent image areas are given a pixel value of all bits set to 0s and the surrounding areas a value of all bits set to 1s.

Bit masking is accomplished by doing bitwise AND-ing in order to clear (which means set to false) a subset of the bits in the byte.

For example, if we have the following mask and the following byte:

Mask: `00001111`
Byte: `01010101`

Applying the mask to the value means that we want to “clear” or zero out the first 4 bits, and keep only the last 4 bits. The result is:

Mask: `00001111`
Byte: `01010101`
Result: `00000101`

In Zoom, this same technique is used to clear (delete) your unmoving background while keeping your head and shoulders!

Now, onto the Lab Requirements!

Clone the **L2: Bytes** repository.

An **Abstract Data Type (ADT)** is a data type (such as a class) that is defined by its operations and behaviors rather than by how these operations are implemented. Users interact with an ADT only through its public interface, which hides the underlying implementation details.

For the first milestone, you will need to write a **HIGH-LEVEL** algorithmic design description or pseudocode to describe the main steps that your program is going to take. See What is an Algorithm? For examples of this.

In this lab, you will be tasked with completing a `Byte` class that does a variety of things related to bytes while remaining an ADT. Code that fails to meet ADT standards will receive substantially reduced credit. The `GetRandom` class is provided for your benefit. You should not be making changes to it.

Inside the `Byte` class, you will be using an array of size 8, which should remain a private member attribute. Sometimes all of the lower bits will be utilized for data, and sometimes, such as when mimicking the magician’s trick, only the lower 7 bits utilized for the actual data. Read through the starter code, run it, and modify it as follows:

1. Fix and remove all of the FIXMEs, including in the `README.md` file.
2. You will also need to implement all of the following:
    a. A new **accessor (getter) method** that returns a single bit from the eight-bit `Byte` array. This method should have one integer parameter between 0 and 7 that represents the index of the

bit desired. Note: this literally means the method RETURNS a bit from the method call- it should not be chatty and so should not also print it.

b. A new **Boolean accessor (getter) method** that checks to see if all of the first 7 bits of the `Byte` object have even parity, returning true if it has even parity and false otherwise. A good name for this might be something like is-even()

c. A **void mutator (setter) method** that sets the final parity bit for even parity. This should be given a good name such as set-even-parity(). Note: This means you will check the first 7 bits and then set the final parity bit to ensure even parity for the entire `Byte`.

d. A **void mutator (setter) method** that "flips" all of the 8 bits of the called `Byte` object, meaning that it turns all trues to falses and all falses to true. Be sure to give it a meaningful name.

e. A **mutator (setter) method** that takes an 8-bit `Byte` object as an input parameter that can be used as a mask. The mask should then be applied to the `Byte` object, changing its data to the "masked" data. Be sure to give this a meaningful name.

f. An **overloaded << operator** so `cout` should be used to print the `Byte` object using `cout` rather than the show method.

g. A main function that fully tests ALL of the methods of your enhanced `Byte` class. (Note that here you are not expected to use unit tests.)

3. Your program must include a descriptive header as a comment at the top of your source code which includes your name as author, the assignment name and purpose, and a statement that this is the work of you as author except where specifically documented.
4. It must use only meaningful variable and method names.
5. It must have a descriptive docstring for each method when the purpose is not clear, and, it should have appropriate pre- and post-conditions whenever these are not clear.
6. Meet all milestones as described in Moodle.

To Submit

1) Commit your milestone versions to GitHub so that we have access to the most recent version of your program.
2) Complete the implementation and the `README.md` file, commit and push your repository to the **main branch** in your Github repository before this project is due, and submit the link to your Github repository in the Moodle textbox. There are more than 1000 repositories in the csc236 Github organization, so do not expect the TAs to be able to find your repository if you fail to submit the link. Also, do not commit or push anything after the deadline or it will be counted as late.