

Extra Credit 01

In the **Programming/ExtraCredit01** folder of your class repository on GitHub, create a file named **NumberBase.h** file. Ensure your file has the proper header guard. Only include: **iostream**, **string**, **cmath**, and **stdexcept**. All following tasks should be in a namespace called **dsec**:

Task 1: (1 point)

Create a class named **NumberBase** that declares the 3 following public member variables:

```
static const string DIGITS;  
static const string OCTALS[8];  
static const string HEXADECIMALS[16];
```

Task 2: (1 point)

Outside of the class, assign the 3 member variables the 3 following values respectively:

```
= "0123456789ABCDEF"  
= {"000", "001", "010", "011", "100", "101", "110", "111"}  
= {"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",  
   "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"};
```

Task 3: (1 point)

Create a public static int method called **getValue()** that has a char parameter. The function should return the decimal value of the character parameter. Use the table below for the proper conversions. If the character is not a valid character, return -1. Letters are case insensitive.

| char | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| int | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

Task 4: (1 point)

Create a public static bool method called **validate()** that has a constant string reference parameter and an integer parameter. The string parameter represents a signed binary, octal, or hexadecimal number. The integer parameter represents the base of the number. If the base is invalid !(2 or 8 or 16) or the string is empty, return false. Return true if every character in the string is valid for the base. 2: (0, 1), 8: (0 - 7), 16: (0 - 9, A - F, a - f)

```
Example: validate("0123", 2) => false, validate("1010", 2) => true  
        validate("678", 8) => false, validate("71450", 8) => true  
        validate("3BALL", 16) => false, validate("ACE5", 16) => true
```

Task 5: (1 point)

Create a public static string method called **arithmeticShiftRight()** that has a constant string reference parameter and two integer parameters. The string parameter represents a signed binary, octal, or hexadecimal number. The second integer parameter represents the base of the number. Return a string that represents the result of the string number arithmetically shifted to right by the number of positions indicated by the first integer parameter. Note: an arithmetic right shift prepends a sign digit to the left of the number on every shift. Validate the string. If the string is invalid, throw an exception.

```
Example: arithmeticShiftRight("01010", 1, 2) => "00101"  
        arithmeticShiftRight("1001", 1, 2) => "1100"  
        arithmeticShiftRight("234751", 1, 8) => "023475"  
        arithmeticShiftRight("54627", 1, 8) => "75462"  
        arithmeticShiftRight("AA1A", 1, 16) => "FAA1"  
        arithmeticShiftRight("45A7", 1, 16) => "045A"
```

Task 6: (1 point)

Create a public static string method called **logicalShiftLeft()** that has a constant string reference parameter and two integer parameters. The string parameter represents a signed binary, octal, or hexadecimal number. The second integer parameter represents the base of the number. Return a string that represents the result of the string number logically shifted to left by the number of positions indicated by the first integer parameter. Note: a logical left shift appends a 0 to the right of the number on every shift. Validate the string. If the string is invalid, throw an exception.

```
Example: logicalShiftLeft("01010", 1, 2) => "10100"
        logicalShiftLeft("1001", 1, 2) => "0010"
        logicalShiftLeft("234751", 1, 8) => "347510"
        logicalShiftLeft("54627", 1, 8) => "46270"
        logicalShiftLeft("AA1A", 1, 16) => "AA1A0"
        logicalShiftLeft("45A7", 1, 16) => "5A70"
```

Task 7: (1 point)

Create a public static string method called **arithmeticShiftLeft()** that has a constant string reference parameter and two integer parameters. The string parameter represents a signed binary, octal, or hexadecimal number. The second integer parameter represents the base of the number. Return a string that represents the result of the string number logically shifted to left by the number of positions indicated by the first integer parameter. Note: an arithmetic left shift appends a 0 to the right of the number on every shift. Validate the string. If the string is invalid, throw an exception.

```
Example: arithmeticShiftLeft("01010", 1, 2) => "10100"
        arithmeticShiftLeft("1001", 1, 2) => "0010"
        arithmeticShiftLeft("234751", 1, 8) => "347510"
        arithmeticShiftLeft("54627", 1, 8) => "46270"
        arithmeticShiftLeft("AA1A", 1, 16) => "AA1A0"
        arithmeticShiftLeft("45A7", 1, 16) => "5A70"
```

Task 8: (1 point)

Create a public static string method called **logicalShiftRight ()** that has a constant string reference parameter and two integer parameters. The string parameter represents a signed binary, octal, or hexadecimal number. The second integer parameter represents the base of the number. Return a string that represents the result of the string number logically shifted to right by the number of positions indicated by the first integer parameter. Note: a logical right shift prepends a 0 to the left of the number on every shift. Validate the string. If the string is invalid, throw an exception.

```
Example: logicalShiftRight("01010", 1, 2) => "00101"
        logicalShiftRight("1001", 1, 2) => "0100"
        logicalShiftRight("234751", 1, 8) => "023475"
        logicalShiftRight("54627", 1, 8) => "05462"
        logicalShiftRight("AA1A", 1, 16) => "0AA1"
        logicalShiftRight("45A7", 1, 16) => "045A"
```

Task 9: (1 point)

Create a public static string method called **onesComplement()** that has a constant string reference parameter and an integer. The string parameter represents a signed binary, octal, or hexadecimal number. The integer parameter represents the base of the number. Return the one's complement of the string parameter. Ensure the string is valid for the base. If the string is invalid, throw an exception.

```
Example: onesComplement("01010", 2) => "10101"
        onesComplement("1000", 2) => "0111"
        onesComplement("2025", 8) => "5752"
        onesComplement("954F", 16) => "6AB0"
```

Task 10: (1 point)

Create a public static string method called **twosComplement()** that has a constant string reference parameter and an integer. The string parameter represents a signed binary, octal, or hexadecimal number. The integer parameter represents the base of the number. Return the two's complement of the string parameter. Note: 1 in binary is -1. You need to add positive 1. Also, the two's complement of a negative power of 2 in binary will result in the original number. Do not return the original number.

```
Example: twosComplement("10101",2) => "01011"
        twosComplement("1000",2) => "01000"
        twosComplement("6547",8) => "1231"
        twosComplement("4A2E",16) => "B5D2"
```

Task 11: (1 point)

Create a public static string method called **add()** that has two constant string reference parameters and an integer parameter. The string parameters represent signed binary, octal or hexadecimal numbers. The integer parameter represents the base of the two numbers. Return a string that represents the signed sum of the two string parameters. Ensure the string parameters are valid for the base. If the strings are invalid, throw an exception. Do not convert the numbers to decimal, perform the addition then convert the sum to the base.

```
Example: add("01010", "01100", 2) => "010110"
        add("23400751", "73476230", 8) => "17077201"
        add("AA1A", "989ACD0B", 16) => "989A7725"
```

Task 12: (1 point)

Create a public static string method called **subtract()** that has two constant string reference parameters and an integer parameter. The string parameters represent signed binary, octal or hexadecimal numbers. The integer parameter represents the base of the two numbers. Return a string that represents the signed difference of the two string parameters. Ensure the string parameters are valid for the base. If the strings are invalid, throw an exception. Do not convert the numbers to decimal, perform the subtraction then convert the sum to the base.

```
Example: subtract("01010", "11110", 2) => "01100"
        subtract("23400751", "73476230", 8) => "27702521"
        subtract("AA1A", "989ACD0B", 16) => "6764DD0F"
```

Task 13: (1 point)

Create a public static string method called **toBase()** that has two integer parameters. The first parameter represents the decimal number that should be converted to the base indicated by the second parameter. Use repeated division on the absolute value of the decimal number. Ensure the result of the repeated division is positive before proceeding. If the result is not positive, prepend a '0' to result. If the decimal number is negative, convert the result to its two's complement form. Return the result. Ensure the base is either 2, 8, or 16.

```
Example: toBase(25,2) => "011001", toBase(-25,2) => "100111"
        toBase(2625,8) => "05101", toBase(-2625,8) => "72677"
        toBase(2625,16) => "0A41", toBase(-2625,16) => "F5BF"
```

Task 14: (1 point)

Create a public static string method called **pad()** that has a constant string reference parameter and two integer parameters. The string parameter represents a signed binary, octal, or hexadecimal number. The second integer parameter represents the base of the number. If the first integer parameter is greater than the length of the string parameter, return a padded version of the string number whose length equals the first integer parameter. The padded value should be that of the sign digit. If the string is invalid, throw an exception.

```
Example: pad("1010", 8, 2) => "11111010", pad("010", 5, 2) => "00010"
        pad("6010", 6, 8) => "776010", pad("334", 6, 8) => "000334"
        pad("CAFE", 5, 16) => "FCAFE", pad("7D5", 6, 16) => "0007D5"
```

Task 15: (1 point)

Create a public static string method called **trim()** that has a constant string reference parameter and an integer parameter. The string parameter represents a signed binary, octal, or hexadecimal number. The integer parameter represents the base of the number. Return a string that represents the string number with any padded and all sign digits removed. Ensure the string is valid for the base. If the string is invalid, throw an exception.

Example: `trim("11111010", 2) => "1010"`, `trim("00000111", 2) => "0111"`
`trim("7770765", 8) => "70765"`, `trim("0000655", 8) => "0655"`
`trim("7775765", 8) => "5765"`, `trim("0002655", 8) => "2655"`
`trim("FFF4CE5", 16) => "F4CE5"`, `trim("0000A", 16) => "0A"`
`trim("FFFACE5", 16) => "ACE5"`, `trim("00067", 16) => "67"`

Task 16: (1 point)

Create a public static string method called **toDecimal()** that takes a constant string reference parameter and an integer parameter. The string parameter represents a signed binary, octal, or hexadecimal number. The integer parameter represents the base of the number. Return an integer that represents the string number as a signed decimal number. Calculate the weighted sum. Afterwards, if the string number is negative for the respective base, subtract the base raised to the power of the number of characters in the string from the weighted sum. If the string is invalid, throw an exception.

$$\begin{aligned}\text{Example: } 1010_2 &= -2^4 + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) \\ &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) - 2^4 \\ &= 8 + 0 + 2 + 0 - 16 \\ &= -6\end{aligned}$$

$$\begin{aligned}\text{Example: } 01010_2 &= (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) \\ &= 0 + 8 + 0 + 2 + 0 \\ &= 10\end{aligned}$$

$$\begin{aligned}\text{Example: } 4321_8 &= -8^4 + (4 \times 8^3) + (3 \times 8^2) + (2 \times 8^1) + (1 \times 8^0) \\ &= (4 \times 8^3) + (3 \times 8^2) + (2 \times 8^1) + (1 \times 8^0) - 8^4 \\ &= 2,048 + 192 + 16 + 8 - 4,096 \\ &= -1,832\end{aligned}$$

$$\begin{aligned}\text{Example: } 12345_8 &= (1 \times 8^4) + (2 \times 8^3) + (3 \times 8^2) + (4 \times 8^1) + (5 \times 8^0) \\ &= 4,096 + 1,024 + 192 + 32 + 5 \\ &= 5,349\end{aligned}$$

$$\begin{aligned}\text{Example: } FACE_{16} &= -16^4 + (F \times 16^3) + (A \times 16^2) + (C \times 16^1) + (E \times 16^0) \\ &= (15 \times 16^3) + (10 \times 16^2) + (12 \times 16^1) + (14 \times 16^0) - 16^4 \\ &= 61,440 + 2,560 + 192 + 14 - 65,536 \\ &= -1330\end{aligned}$$

$$\begin{aligned}\text{Example: } 3BA17_{16} &= (3 \times 16^4) + (B \times 16^3) + (A \times 16^2) + (1 \times 16^1) + (7 \times 16^0) \\ &= (3 \times 16^4) + (11 \times 16^3) + (10 \times 16^2) + (1 \times 16^1) + (7 \times 16^0) \\ &= 196,608 + 45,056 + 160 + 16 + 7 \\ &= 241,847\end{aligned}$$

Task 17: (1 point)

Create a public static string method called **toBinary()** that takes a constant string reference parameter and an integer parameter. The string parameter represents a signed octal, or hexadecimal number. The integer parameter represents the base of the string number. Return the octal or hexadecimal string number converted to a binary string number. If the string is invalid, throw an exception.

Example: `toBinary("2025", 8) => "010000010101"`
`toBinary("954F", 16) => "1001010101001111"`

Task 18: (1 point)

Create a public static string method called **fromBinary()** that takes a constant string reference parameter and an integer parameter. The string parameter represents a signed binary number. The integer parameter represents the base to which the string parameter should be converted to. Return the octal or hexadecimal conversion. Ensure the string is valid for a binary number and the base parameter is either 8 or 16. If the string or base is invalid, throw an exception.

Example: `fromBinary("110001100111", 8) => "6147"`
`fromBinary("1101011111000000", 16) => "D7C0"`

Task 19: (1 point)

Create a public static string method called **multiply()** that takes two constant string reference parameters and an integer parameter. The string parameters represent signed binary, octal or hexadecimal numbers. The integer parameter represents the base of the numbers. Return a string that represents the signed product of the two string parameters. If the strings are invalid for the base, throw an exception. If the base is not binary, convert the string numbers to binary. At the end of the program, convert the product back to the base. Tip: After making Q and M the same bit length, extend (pad) the two numbers by one. This will handle any overflow concerns.

Example: `multiply("01010", "11110", 2) => "11111101100"`
`multiply("23400751", "73476230", 8) => "7765034155130"`
`multiply("AA1A", "989ACD0B", 16) => "0000022B17ED7211E"`

Task 20: (1 point)

Create a public static string method called **divide()** that takes two constant string reference parameters and an integer parameter. The string parameters represent signed binary, octal or hexadecimal numbers. The integer parameter represents the base of the numbers. Return a string that represents the signed quotient and remainder of the first string divided by the second. The return string should be the concatenation of A and Q in the following format: **"remainder quotient"**. Be aware of the space between A and Q. (**A** + ' ' + **Q**) If the strings are invalid for the base, throw an exception. If the base is not binary, convert the string numbers to binary. At the end of the program, convert the quotient (Q) and remainder (A) back to the base indicated by the integer parameter. Tip: After making Q's bit length greater than or equal to the bit length of M, extend (pad) the two numbers by one. This will handle any overflow concerns.

Example: `divide("01010", "11110", 2) => "000000 111011"`
`divide("234751", "734230", 8) => "0016111 7777774"`
`divide("AA1A", "989ACD0B", 16) => "FFFFFFAA1A 00000000"`
`divide("989ACD0B", "AA1A", 16) => "FEF49 000013425"`

NOTE: There is a space between the remainder and quotient.
Green = quotient, Red = Remainder