```
//ICSI 333. System Fundamentals
//Peter Buonaiuto: 001497626
//TA Omkar Kulkarni
//Spring 2022

//This program will take a user given expression consisting of integers
//and basic operators, and print the strict left to right evaluation of the
//expression; ignoring precedence.
#include <stdio.h> //Preprocessor includes IO functionality
#include <stdbool.h>//Instruction to include boolean functionality
#include <ctype.h>
#include <string.h>
#include <math.h>

    //Global Constants
    const int MAX_EXP_LEN = 80; //The maximum size for the char array
    const int ZERO = 0; //Zero to avoid magic number
    const int ONE = 1; //One to avoid magic number

    //Function definitions
    int strictlyEvaluate(char[]); //Will evaluate a given expression strictly and return result.
    void convertRadix(int, int); //Will convert and print to a new radix

//Main function will prompt the user for an expression, take the given expression and print
the resulting integer.
//Main function calls strictlyEvaluate and convertRadix.
int main() {

    //Define a char array for the expression
    char expression[MAX_EXP_LEN];

    //Prompt the user for the expression, and scan it with whitespaces.
    printf("Enter an expression: "); fflush(stdout);
    scanf("%[^\n]s", expression);
```

```
	//Find and store the result, then print it
	int result = strictlyEvaluate(expression);
	printf("Decimal Answer: %d\n", result); fflush(stdout);



	//TASK 2
	int radix; //The radix to convert to

	printf("What radix would you like to convert to?: "); fflush(stdout);
	scanf("%d", &radix);
	convertRadix(result, radix);

	return ZERO;
}

//Function which will evaluate a given expression which satisfies the given preconditions.
//Given expression is an array of characters. Whitespace will be ignored. We will treat each
//even index entry as an operand, and each odd indexed entry as an operator.
int strictlyEvaluate(char expression[])
{
	//Remove whitespace for proper indexing and evaluation
	int len = ZERO; //Keep track of the length of the new string, for index handling
	for (int i = ZERO; expression[i]; i++)
		if (expression[i] != ' ')  //If the token is NOT a whitespace,
		{                          //then we can include this in the adjusted array.
			expression[len] = expression[i];
			len++; //Incriment length once our final array gains a member.
		}
	expression[len] = '\0'; //Denote where the string terminates.



	//Now, begin to evaluate the whitespace-free expression, until its end
	int result = ((int)expression[ZERO] - '0'); //We start with the first operand.

	char* token; //Denotes the token of the expression currently used
```

```
bool isOperator = true; //Alternates to determine when to compute
char operator; //Holds the last read operator in memory for use.

//Iterate over tokens in the expression until the terminal character.S
for (token = &expression[ONE]; *token != '\0'; token++)
{
        if (isOperator)
        {//This token is an operator, save it for next iteration so we know
         //which operation to apply to the next token which is an operand.
              operator = *token;
        }
        else
        {//This token is an operand. Apply the operator with this and the result. Based
        on the operator, we will apply proper function
        //on the result as operand 1 and the current token as operand 2.
        if (operator == '+')
                result += ((int)*token - '0');

        else if (operator == '-')
                result -= ((int)*token - '0');

        else if (operator == '*')
                result *= ((int)*token - '0');

        else
                result /= ((int)*token - '0');
    }

    //Alternate the boolean value to denote the presence of an operator
    //We know that if it was false it is now true, by the preconditions.
    isOperator = !isOperator;
    }

//print the final answer as an integer.
return result;
```

```
}

//Function which converts the given result to the given radix, then prints it.
void convertRadix(int result, int radix)
{
    int remaindr; //Remainder from each division iteration will be used to determine the
    next character in the result.
    bool negative; //Used for displaying negative numbers, while avoiding computing a
    non-existent logarithm.

    //The values which will correspond to a given remainder, giving the final resulting value
    in the answer.
    char values[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};

    //If the result was negative, make it positive and add the
    //negative later. Necessary as log is only defined for x > 0.
    negative = (result < ZERO);
    result = (negative) ? (-result) : result; //Absolute value

    //The length of the final result, determined by predicting the
    //number of remainders we will observe until the quotient is 0.
    int new_length = (result == ZERO) ? ONE : ((int)(log(result)/log(radix)) +ONE);
    char new_answer[new_length]; //Array which will hold the new answer
    int index = ZERO; //Index to determine where in the new list the new value will go,
    counting iterations of the do while.


    //Main conversion algorithm:
        //Until the quotient is zero, divide it by the radix.
        //It's remainder in each iteration will be the next character in the result.
        //We increment the index AFTER accessing the value using index++.
    do {
        remaindr = result % radix;
        result /= radix;
        new_answer[index++] = values[remaindr];
```

```
        } while (result != ZERO);

        printf("Answer in base %d is %c", radix, (negative ? '-' : '\0'));
        fflush(stdout);

        //Result print loop:
            //Beginning at the second to last index (we do not want '\0'),
            //Print each element so that we read the result in backward order.
            //This is because the algorithm finds the final char, first.
        for (int i = --index; i>=ZERO; i--)
        {
            printf("%c", new_answer[i]);
            fflush(stdout);
        }
        printf("\n"); fflush(stdout);

        return;
}
```