Interpreter code description:

1. Data structures used

In order to implement the interpreter we chose to work with arrays as data structures to reference the memory. Following are the reasons behind our choice:

- Our memory will consist of elements of the same data type. We chose arrays as they allow grouping elements of the same data type and manipulating them easily.
- As indexing is done in O(1) using arrays and as we will be doing a lot of memory operations that require accessing specific locations of the memory such as referring to an operand in the memory or jumping to a specific label, using arrays is our best alternative.

2. Functions

Following the same reasoning, we decided to divide the code into multiple functions to help the reader better understand our code and to make clear distinctions between parts of the code that should be handled separately.

- loadData()

  → This function is used to load data from the data part of the code to the memory. It reads line by line from a file that contains code in machine language as an integer and takes the last 4 digits of the instruction and puts it in the data memory. We take the last 4 digits as they are the ones that hold the value of the variable.
  This function takes no input arguments since it makes use of the global variable dataMemory.

- loadProgram()

  → This function is used to load instructions from the program part of the code to the memory. It reads line by line from the file that was used in loadData(). It reads the line as a string and not an integer. The reason behind this is that if the instruction starts with zeros, they will not be taken into consideration as they will be considered to be leading zeros.
  This function takes no input as it uses the global variable programMemory.
  This function returns the number of lines of the program part which will be used to control the while loop.

3. How to run the code

This interpreter takes as input the output file of the assembler which again contains code written in machine language. The interpreter runs the machine code and produces the expected output of the program.
To run the code, the output file of the assembler should be created and filled, by the assembler, in the same location as the interpreter.
The code can be run on any IDE provided that a GCC compiler is installed.

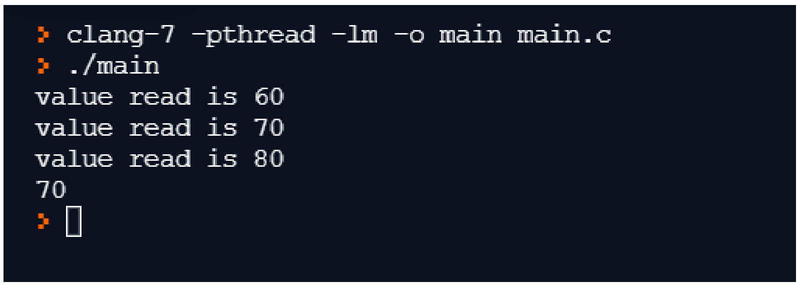
4. Sample Input and Output

1. Simple Code

- Machine code in input file:

```
+000000000
+000010000
+000020000
+000030000
+9999999999
+7300000000
+7300010000
+7300020000
+0300000000
+1100010001
+0300010000
+1100020002
+0000030000
-2100020003
-7300030000
+9000000000
+9999999999
+000000060
+000000070
+000000080
```

- Expected output of the program (could be printing to the screen):



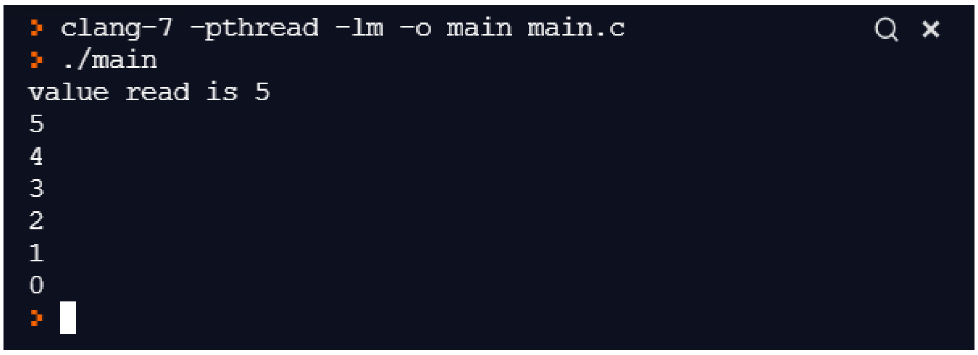
```
> clang-7 -pthread -lm -o main main.c
> ./main
value read is 60
value read is 70
value read is 80
70
> 
```

2. **Medium Complexity Code**

- Machine code in input file:

```
+000000000
+9999999999
+7300000000
-5100000007
-7300000000
+0000010000
-1100000000
+0000000000
+5100000002
+9000000000
+9999999999
0000000005
```

- Expected output of the program (could be printing to the screen):



```
> clang-7 -pthread -lm -o main main.c
> ./main
value read is 5
5
4
3
2
1
0
>
```

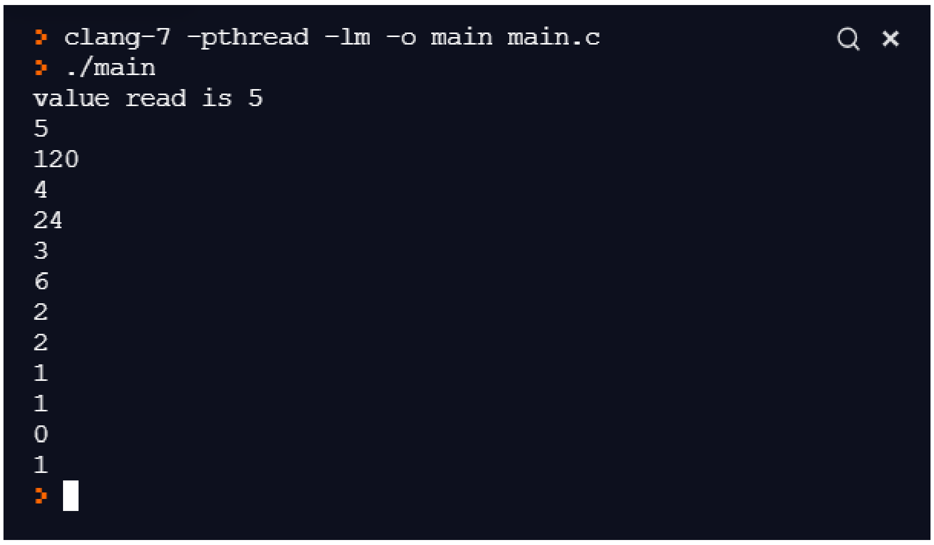
3. **High Complexity Code**

- Machine code in input file:

```
+000000000
+000010001
+000020000
+9999999999
+7300000000
-5100000016
-7300000000
+4100000010
+0100000002
+0300020000
+2100010001
```

+0000010000
-1100020002
+5100020005
-7300010000
+0200010001

- Expected output of the program (could be printing to the screen):



```
> clang-7 -pthread -lm -o main main.c
> ./main
value read is 5
5
120
4
24
3
6
2
2
1
1
0
1
> 
```