```c
/*
    Jeffrey Lansford
    Program 1
    9/1/2020
    Input: no input
    Output:
        JEFFREY LANSFORD
        HELLO
        WORLD
        memory locations main,foo,A,B,C,D, and S
    Preconditions: None
    Postconditions: None
*/
#include <stdio.h>
#include <stdlib.h>


// function todo 256^x
int square_256(int i)
{
    int square = 1;
    int x;
    for (x = 0; x < i; x++)
    {
        square = square * 256;
    }
    return square;
}


// global variable for memory location test
```

```
int B = 20;


// function for memory test
int foo()
{
    int E[20];
}


int main()
{
    // Integer Array to store bytes of integers
    static int A[100];
    // Pointer to print integer array as string
    char *S;
    // Fill array with my name and hello for testing 0 byte
    A[0] = 'J' + ('E' * square_256(1)) + ('F' * square_256(2)) + ('F' * square_256(3));
    A[1] = 'R' + ('E' * square_256(1)) + ('Y' * square_256(2)) + (' ' * square_256(3));
    A[2] = 'L' + ('A' * square_256(1)) + ('N' * square_256(2)) + ('S' * square_256(3));
    A[3] = 'F' + ('O' * square_256(1)) + ('R' * square_256(2)) + ('D' * square_256(3));
    A[4] = '\n' + ('H' * square_256(1)) + ('E' * square_256(2)) + ('L' * square_256(3));
    A[5] = 'L' + ('O' * square_256(1)) + ('\n' * square_256(2)) + ('W' * square_256(3));


    // Line for testing 0 byte
    // A[5] = 'L' + (0 * square_256(1)) + ('O' * square_256(2)) + ('\n' * square_256(3));


    // fill rest of array to unsure test has worked
    A[6] = 'O' + ('R' * square_256(1)) + ('L' * square_256(2)) + ('D' * square_256(3));


    // fills element to make sure that it does not print anything else
```

```
    A[7] = 0;


    // sets char pointer to integer array
    S = A;
    // prints as char array
    printf("%s\n", S);
    // test for memory location of variables
    char *C = malloc(100);
    static int D[10];
    printf("Location of       main is %20u\n", main);
    printf("Location of        foo is %20u\n", foo);
    printf("Location of          B is %20u\n", &B);
    printf("Location of          D is %20u\n", D);
    printf("Location of         *C is %20u\n", C);
    printf("Location of          S is %20u\n", &S);
    printf("Location of          A is %20u\n", A);
    foo();
}
```

```
jelansfo@porthos:~/CS471/Program1> make
gcc name.c -w -o name
jelansfo@porthos:~/CS471/Program1> ./name
JEFFREY LANSFORD
HELLO
WORLD
Location of       main is              4195786
Location of        foo is              4195779
Location of          B is              6295616
Location of          D is              6296096
Location of         *C is             22685296
Location of          S is            928249488
Location of          A is              6295680
jelansfo@porthos:~/CS471/Program1>
```

a) The array in in the Stack segment of memory. We can proof this by printing the memory locations of the array to see where it is stored in memory. We can print main() and the foo() functions to show the code segment which are the smallest numbers compared to the rest. We can print the memory location in global variable, B, to show the data segment. This memory location should be a lot greater than our code segment locations. Then we can do a malloc() to provide the heap section in memory, that is greater than data segment locations. Then we can print an array in main() to show the stack section of memory that should be far from the heap locations as the stack starts at the top and the heap starts from the bottom. We can see that A's memory location is very far apart from the heap array C, which tells us that it is in the stack section in memory.

```
Location of      main is            4195786
Location of       foo is            4195779
Location of         B is            6295616
Location of        *C is           23676528
Location of         D is           67886288
Location of         S is           67886328
Location of         A is           67886336
jelansfo@porthos:~/CS471/Program1>
```

b) Again with the test, we can see that the pointer to the array is located in the stack segment in memory. We can print the location of the pointer S and see that it is next to A which is in the stack.

c) We can make our array be in another segment in memory by making it a static variable or a global variable and that would put it in the data segment of memory. When can add static to A and that would make it next to our global variable B and static variable D.

```
Location of      main is            4195786
Location of       foo is            4195779
Location of         B is            6295616
Location of         D is            6296096
Location of        *C is           10126960
Location of         S is          382736304
Location of         A is            6295680
```

d) The endianness of my computer is little endian as it is an Intel-based platform which uses little endian style.

e) There are some advantages between the two endianness styles which could have led to them being separate styles and not every company going with one style. Little endian can be easily written with multiple precision math routines as the way it is read in memory is the same as the actual number. They have a 1:1 ratio. Big endian can be easily written to check if the numbers are negative as the most significant bit is first, so we would only have to look at one location. I do not believe that one is better than the other as the advantages and not very significant compared to modern machines speed.

https://thebittheories.com/little-endian-vs-big-endian-b4046c63e1f2

https://en.wikipedia.org/wiki/Endianness

4) We do not need to fill the entire last integer with zero. We can just fill in one byte with zero for the print to stop. We can show with the 'HELLO' segment of the array where we stick a 0 byte between the 'L' and 'O'

```
// Fill array with my name and hello for testing 0 byte
A[0] = 'J' + ('E' * square_256(1)) + ('F' * square_256(2)) + ('F' * square_256(3));
A[1] = 'R' + ('E' * square_256(1)) + ('Y' * square_256(2)) + (' ' * square_256(3));
A[2] = 'L' + ('A' * square_256(1)) + ('N' * square_256(2)) + ('S' * square_256(3));
A[3] = 'F' + ('O' * square_256(1)) + ('R' * square_256(2)) + ('D' * square_256(3));
A[4] = '\n' + ('H' * square_256(1)) + ('E' * square_256(2)) + ('L' * square_256(3));
// A[5] = 'L' + ('O' * square_256(1)) + ('\n' * square_256(2)) + ('W' * square_256(3));

// Line for testing 0 byte
A[5] = 'L' + (0 * square_256(1)) + ('O' * square_256(2)) + ('\n' * square_256(3));

// fill rest of array to unsure test has worked
A[6] = 'O' + ('R' * square_256(1)) + ('L' * square_256(2)) + ('D' * square_256(3));

// fills element to make sure that it does not print anything else
A[7] = 0;
```

And should give:

JEFFREY LANSFORD
HELL

And the result is:

```
jelansfo@porthos:~/CS471/Program1> make
gcc name.c -w -o name
gcc foo.c -w -o foo
jelansfo@porthos:~/CS471/Program1> ./name
JEFFREY LANSFORD
HELL
```