What does this C code do?

```c
int foo(char *s) {
    int L = 0;
    while (*s++) {
        ++L;
    }
    return L;
}
```

Note: Lab 7 is an individual lab.

Also, please read MIPS style guide.

1. Handout

Exam 2.2
Pointers, the Spiral Rule, and Structs

- How to read C type declarations
- C Strings
  - ASCII and null-termination
- Array Indexing vs. Pointers
  - Pointer arithmetic, in particular
- Structs
  - Non-homogenous arrays
  - Padding
Representing strings

- A C-style string is represented by an array of bytes.
  - Elements are one-byte **ASCII codes** for each character.
Strings in C are terminated by the null character (0)

- For example, “Harry Potter” can be stored as a 13-byte array.
2-dimensional arrays in C are laid out in memory as one big array

- E.g., `int A[100][200]` is essentially `int A[20000]`
- "row major order" = rows are laid out contiguously
  - A[i][j+1] comes right after A[i][j]
  - A[i+1][0] comes right after A[i][199]
  - `&A[i][j] = &A[0][0] + ((i * 200) + j) * sizeof(int)`
Array Indexing Implementation of strlen

Which of the following lines of code correctly loads the contents of `string[len]` into `$t0`, assuming that `len` is stored in `$v0`?

```c
int strlen(char *string) {
    int len = 0;
    while (string[len] != 0) {
        len ++;
    }
    return len;
}
```

a) `lb $t0, $v0($a0)`  
b) `add $t0, $a0, $v0`  
   `# & string[len]`

c) Both (a) and (b)  
d) Neither (a) nor (b)
Convert the C code into MIPS assembly

```c
int strlen(char *string) {
    int len = 0;
    while (string[len] != 0) {
        len ++;
    }
    return len;
}
```

```mips
add   $v0, $0, $a0      # len = 0

strlen:
    li    $v0, 0     # len = 0

strlen_loop:
    add   $t0, $a0, $v0        # &string[len]
    lb    $t1, 0($t0)        # index
    beq   $t1, $0, strlen_done
    add   $v0, $v0, 1       # len ++
    j     strlen_loop

strlen_done:
    jr    $ra

jal   strlen
```
Assembly coding can help you gain a better understanding of pointers

```c
int strlen(char *string) {
    int len = 0;
    while (string[len] != 0) {
        len ++;
    }
    return len;
}
```

```c
int strlen(char *string) {
    int len = 0;
    while (*string != 0) {
        string ++;
        len ++;
    }
    return len;
}
```
A pointer is an address

- Two pointers that point to the same thing hold the same address
- Dereferencing a pointer means loading from the pointer’s address

```c
char *char1 = 0x4001
char *char2 = 0x4001
int *x = 0x4004

if (char1 == char2)
{
    *char1 // 'c'
    0x00004000
    0x00004001
    0x00004002
    0x00004003
    0xFF
    0x00004004
    0x00004005
    0x00004006
    0x00004007
}
```

Legend:
- `lb/lbu or sb` for loading 1 byte
- `lw or sw` for loading 4 bytes
Opcode to use depends on pointer type and usage

- Use load/store byte (lb/sb) for char *
- Use load/store half (lh/sh) for short *
- Use load/store word (lw/sw) for int *
- Use load/store single precision floating point (l.s/s.s) for float *

- **Load:** If you need to de-reference pointer to evaluate expression:
  - ... = ... + *p + ...  
  - or- 
  - A[*p]

- **Store:** If it where you put the result of the expression:
  - *p = ...

Pointer arithmetic is useful for pointers to arrays

- Incrementing a pointer (i.e., ++) makes it point to the next element
- The amount added to the pointer depends on the type of pointer
  - pointer = pointer + sizeof(pointer’s type)
  - 1 for char *, 4 for int *, 4 for float *, 8 for double *

char string[4] = {'c', 'h', 'a', 'r'};
int array[2] = {-7, 9};

char *cp = string;
cp += j

int *ip = array;
ip += j
Convert the C code to MIPS assembly to understand what is going on

```c
int strlen(char *string) {
    int len = 0;
    while (*string != 0) {
        string ++;
        len ++;
    }
    return len;
}
```

```assembly
strlen:
    li $v0, 0 # len: $v0
strlen_loop:
    lb $t0, $a0 ($a0) # *string
    beq $t0, $0, strlen_done
    add $a0, $a0, 1 # string +
    add $v0, $v0, 1 # len +
strlen_loop
strlen_done:
    jr $ra
```
i>clicker

Suppose I modified the C code to an integer array from a string.

```c
int numNotZero(int *array) {
    int len = 0;

    while (*array != 0) {
        array ++;
        len ++;
    }

    return len;
}
```

Which of the following lines of code would correctly execute the instruction `array ++`?

a) `add $a0, $a0, 1`

b) `add $a0, $a0, 2`

c) `add $a0, $a0, 4`

d) The C code’s behavior is undefined
Compilers/assemblers insert padding to “naturally align” data in structs

- Structs are like arrays, but the elements can be different types.
  - Same with objects

- Sometimes you can reorganize fields to eliminate padding.
- Structs must align to the largest data type ≥ 4B

```c
struct {
    int a;
    char b;
    short c[4];
    int d;
}  // 4B → address % 4 == 0
```
How big is this structure?

```
struct {
    char c;
    char *c_ptr[4];
}
```

a) 4 bytes  
b) 5 bytes  
c) 8 bytes  
d) 17 bytes  
e) 20 bytes
Summary

- Pointers are just addresses!!
  - “Pointees” are locations in memory
- Pointer arithmetic updates the address held by the pointer
  - “string ++” points to the next element in an array
  - Pointers are typed so address is incremented by sizeof(pointee)