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.

<table>
<thead>
<tr>
<th>char</th>
<th>ASCII code</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>33</td>
</tr>
<tr>
<td>&quot;</td>
<td>34</td>
</tr>
<tr>
<td>#</td>
<td>35</td>
</tr>
<tr>
<td>$</td>
<td>36</td>
</tr>
<tr>
<td>%</td>
<td>37</td>
</tr>
<tr>
<td>&amp;</td>
<td>38</td>
</tr>
<tr>
<td>'</td>
<td>39</td>
</tr>
<tr>
<td>(</td>
<td>40</td>
</tr>
<tr>
<td>)</td>
<td>41</td>
</tr>
<tr>
<td>*</td>
<td>42</td>
</tr>
<tr>
<td>+</td>
<td>43</td>
</tr>
<tr>
<td>,</td>
<td>44</td>
</tr>
<tr>
<td>-</td>
<td>45</td>
</tr>
<tr>
<td>.</td>
<td>46</td>
</tr>
<tr>
<td>/</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>113</td>
</tr>
<tr>
<td></td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>118</td>
</tr>
<tr>
<td></td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>del</td>
</tr>
</tbody>
</table>
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., \texttt{int A[100][200]} is essentially \texttt{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)

\[
\begin{array}{cccc}
\end{array}
\]
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.

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

```mips
add: $v0, $0, $0

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

```
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, $t1   # len ++
  j strlen_loop

strlen_done:
  jr $ra
  jal strlen
```
Assembly coding can help you gain a better understanding of pointers

```c
char string[20]

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

```mips
strlen:
    li $v0, 0  # len: $0

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

strlen_done:
    jr $ra
```
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
Clockwise/Spiral Rule: Parse any C declaration in your head!


Starting with the unknown element, move in a spiral/clockwise direction; when encountering the following elements replace them with the corresponding English statements:

1. `[X]` or `[]` => Array X size of... or Array undefined size of...
2. `(type1, type2)` => function passing type1 and type2 returning...
3. `*` => pointer(s) to...

Keep doing this in a spiral/clockwise direction until all tokens have been covered.

Always resolve anything in parenthesis first!

```
char *str[10];
```
More Examples (Arrays and Pointers)

\[
\text{int } *x[]; \quad x \text{ is an array of pointers to integers.}
\]

\[
\text{int } (*y)[]; \quad y \text{ is a pointer to an array of integers.}
\]
More Examples (Const and Pointers)

- const char *chptr;
- char * const chptr;

chptr is ...  
- a) A character that points to a constant  
- b) A pointer to a char  
- c) A constant pointer to a char  
- d) A pointer to a constant char

\[
\begin{align*}
\text{const char } & c2 = 'y'; \quad \text{\textcolor{green}{\checkmark}} \\
& *\text{chptr} = 'x'; \quad \textcolor{red}{\times} \\
& \text{chptr} = & c2; \quad \textcolor{green}{\checkmark}
\end{align*}
\]

\[
\begin{align*}
& \text{chptr} = & c2; \quad \textcolor{red}{\times} \\
& *\text{chptr} = 'y'; \quad \textcolor{green}{\checkmark}
\end{align*}
\]
More Examples (Functions and Pointers)

z is a function that takes an integer and returns a ptr to an integer.

\begin{verbatim}
q = $f(z)$;
\end{verbatim}

q is ...

a) A pointer to an integer 

b) A pointer to an integer that is multiplied with a different integer 

c) A function that takes an integer and returns a pointer to an integer 

d) A function that takes a pointer to an integer and returns an integer 

e) A pointer to a function that takes an integer and returns an integer
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 $\geq 4 \times 8$

```c
struct {
    int a;
    char b;
    short c[4];
    int d;
}  // 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)