What does this C code do?

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

Lab 7 is an individual lab.

Read the style guide.
Informal Early Feedback debrief

- Let’s talk about why the course is structured the way it is
  - Frequent exams
  - Clickers
  - Discussion sections
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.
  - A 0 value marks the end of the array.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>space</td>
<td>48</td>
<td>0</td>
<td>64</td>
<td>@</td>
<td>80</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
<td>49</td>
<td>1</td>
<td>65</td>
<td>A</td>
<td>81</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>50</td>
<td>2</td>
<td>66</td>
<td>B</td>
<td>82</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>51</td>
<td>3</td>
<td>67</td>
<td>C</td>
<td>83</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td>52</td>
<td>4</td>
<td>68</td>
<td>D</td>
<td>84</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
<td>53</td>
<td>5</td>
<td>69</td>
<td>E</td>
<td>85</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
<td>54</td>
<td>6</td>
<td>70</td>
<td>F</td>
<td>86</td>
</tr>
<tr>
<td>39</td>
<td>\</td>
<td>55</td>
<td>7</td>
<td>71</td>
<td>G</td>
<td>87</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>56</td>
<td>8</td>
<td>72</td>
<td>H</td>
<td>88</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td>57</td>
<td>9</td>
<td>73</td>
<td>I</td>
<td>89</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td>58</td>
<td>:</td>
<td>74</td>
<td>J</td>
<td>90</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
<td>59</td>
<td></td>
<td>75</td>
<td>K</td>
<td>91</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
<td>60</td>
<td>&lt;</td>
<td>76</td>
<td>L</td>
<td>92</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
<td>61</td>
<td>=</td>
<td>77</td>
<td>M</td>
<td>93</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
<td>62</td>
<td>&gt;</td>
<td>78</td>
<td>N</td>
<td>94</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
<td>63</td>
<td>?</td>
<td>79</td>
<td>O</td>
<td>95</td>
</tr>
<tr>
<td>96</td>
<td>`</td>
<td>112</td>
<td>p</td>
<td>97</td>
<td>a</td>
<td>113</td>
</tr>
<tr>
<td>114</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>b</td>
<td>115</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>c</td>
<td>116</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>d</td>
<td>117</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>e</td>
<td>118</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>f</td>
<td>119</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>g</td>
<td>120</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>h</td>
<td>121</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>i</td>
<td>122</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>j</td>
<td>123</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>k</td>
<td>124</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>l</td>
<td>125</td>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>124</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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.

```
72 97 114 114 121 32 80 111 116 116 101 114 0
Harry Potter \0
```

Char string [13]

String [5-]
Array Indexing Implementation of strlen

Which of the following lines of code correctly loads the contents of string[len] into $t0, assume 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
   lb $t0, 0($t0)

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;
}
```

```assembly
_strlen:
    li $v0, 0 # len = 0

loop:
    add $t0, $a0, $v0 # & string[len]
    lb $t0, 0($t0)
    beq $t0, $0, done_loop

    add $v0, $v0, 1 # len++

    j loop # 3

done_loop:
    jr $ra
```
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;
}
```
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

char &c = *char1;
char &h = *char2;
int &r = *x;
```
Opcode to use depends on pointer type and usage

- Use load/store byte (lb/sb) for char * - 1 byte
- Use load/store half (lh/sb) for short * - 2 bytes
- Use load/store word (lw/sw) for int * - 4 bytes
- Use load single precision floating point (l.s/s.s) for float *

**Load:** If you need to de-reference pointer to evaluate expression:
- \( \ldots = \ldots + \ast p + \ldots \) or \( A[\ast p] \)

**Store:** If it is where you put the result of the expression:
- \( \ast p = \ldots \)
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 *

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

<table>
<thead>
<tr>
<th>CP</th>
<th>0x00004000</th>
<th>0x00004001</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'c'</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>0x00004004</td>
<td>0x00004005</td>
</tr>
<tr>
<td></td>
<td>'a'</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>0x00004006</td>
<td>0x00004007</td>
</tr>
<tr>
<td></td>
<td>'r'</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>0x00004008</td>
<td>0x00004009</td>
</tr>
<tr>
<td></td>
<td>0xF9</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>0x0000400A</td>
<td>0x0000400B</td>
</tr>
<tr>
<td></td>
<td>0x09</td>
<td>0x00</td>
</tr>
<tr>
<td>IP</td>
<td>0x0000400C</td>
<td>0x00</td>
</tr>
<tr>
<td></td>
<td>0x00</td>
<td>0x00</td>
</tr>
</tbody>
</table>

```

```
```
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
str_len:
    add $v0, $s0, 1
    add $v0, $v0, 1
    j loop

loop:
    lb $t0, 0($a0)
    add $t0, $t0, 1
    add $v0, $v0, 1
    j loop

done_loop:
    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
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)
Compilers/assemblers insert padding to “naturally align” data in structs

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

```c
struct st {
    int a;       // -4
    char b;      // -1
    short c[4];  // -2 (4)
    char *d;     // -4

    padding
}
```
Rearranging struct elements can remove padding

```c
struct st2 {
    int a;
    char *d;
    short c[4];
    char b;
}
```
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
Arrays of structs must align with the largest element in the struct

```c
struct st2 {
    int a;
    char *d;
    short c[4];
    char b;
}

st2 struct_array2[2];
```

What address goes here?  

A) 17  
B) 18  
C) 20  
D) 32  
E) 40
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. \([X] \text{ or } []\) => Array X size of... or Array undefined size of...
2. \((\text{type1, type2})\) => 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)

```c
int *x[];
array of pointers to integers

int (*y)[];
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
More Examples (Functions and Pointers)

\[ \text{int } *z(\text{int}); \]

\[ \text{int } (*q)(\text{int}); \]

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
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)