# What does this C code do?

```
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

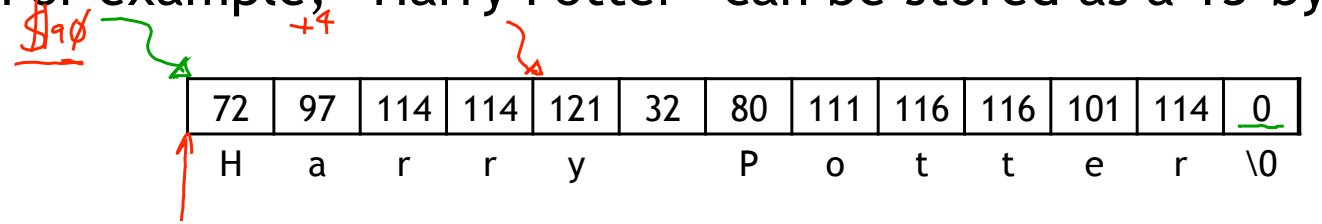
char

- A C-style string is represented by an array of bytes.
  - Elements are one-byte ASCII codes for each character.

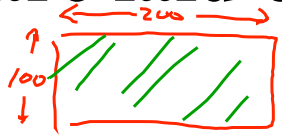
| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | space | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p |
| 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | ” | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | $ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | ’ | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | ( | 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 | ) | 57 | 9 | 73 | I | 89 | Y | 105 | I | 121 | y |
| 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | + | 59 | ; | 75 | K | 91 | [ | 107 | k | 123 | { |
| 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | \| |
| 45 | - | 61 | = | 77 | M | 93 | ] | 109 | m | 125 | } |
| 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | del |

# 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)

&A[i][j] =

&A + (i * 200 + j) * 4

sizeof(int)

| | | | |
|---|---|---|---|
| A[0][0] | A[0][1] | A[0][2] | A[0][3] |
| A[1][0] | A[1][1] | A[1][2] | A[1][3] |
| A[2][0] | A[2][1] | A[2][2] | A[2][3] |
| A[3][0] | A[3][1] | A[3][2] | A[3][3] |

| | |
|---|---|
| &A | A[0][0] |
| &A+4 | A[0][1] |
| &A+8 | A[0][2] |
| &A+12 | A[0][3] |
| &A+16 | A[1][0] |
| &A+20 | A[1][1] |
| &A+24 | A[1][2] |
| &A+28 | A[1][3] |
| &A+32 | A[2][0] |
| &A+36 | A[2][1] |
| &A+40 | A[2][2] |
| &A+44 | A[2][3] |
| &A+48 | A[3][0] |
| &A+52 | A[3][1] |
| &A+56 | A[3][2] |
| &A+60 | A[3][3] |

# 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;
}
```

$a0

$v0

a) `lb $t0, $v0($a0)`

const

b)
```
add $t0, $a0, $v0
lb  $t0, 0($t0)
```

# &string[len]
# string[len]

c) Both (a) and (b)

d) Neither (a) nor (b)

# Convert the C code into MIPS assembly

```
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)
    beq  $t1, $0, strlen_done
    add  $v0, $v0, 1     # len ++
    j    strlen_loop
strlen_done:
    jr   $ra
```

```
add: $v0, $0, $0
jal strlen
```

# Assembly coding can help you gain a better understanding of pointers

char string[20]

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

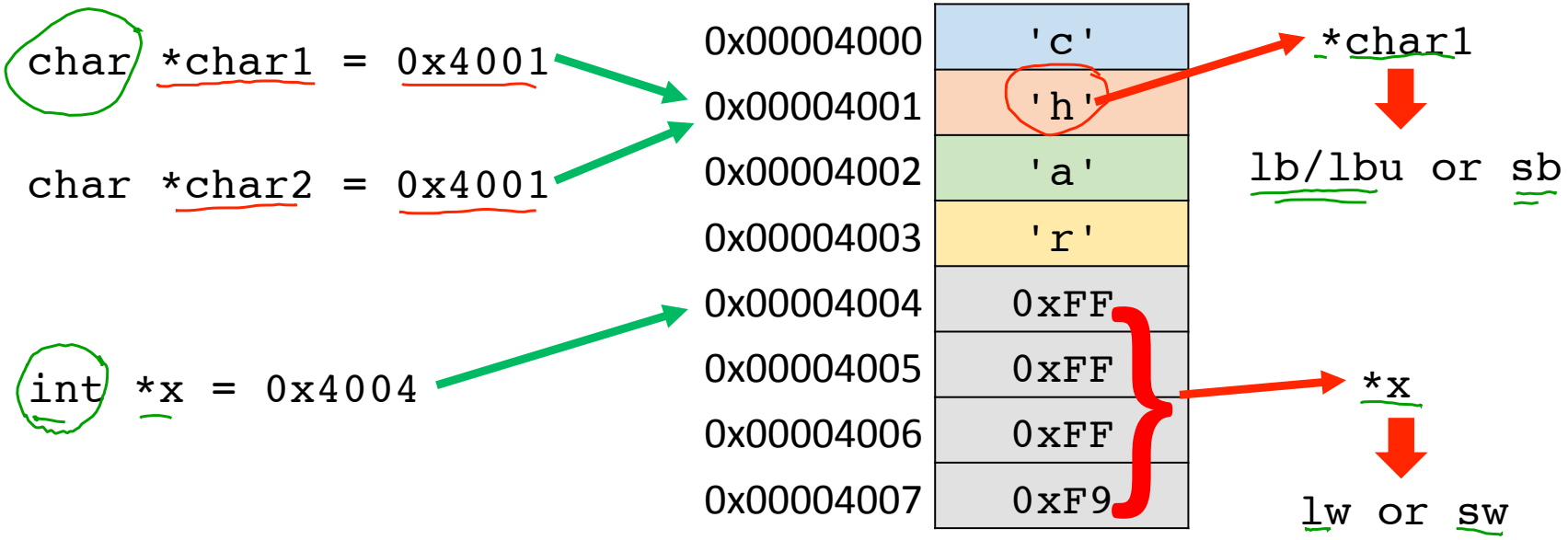
```
int strlen(char *string) {
  int len = 0;
  while (*string != 0) {
    string ++;
    len ++;
  }
  return len;
}
```

ptr dereference

pointer arithmetic

# 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



# 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 = …

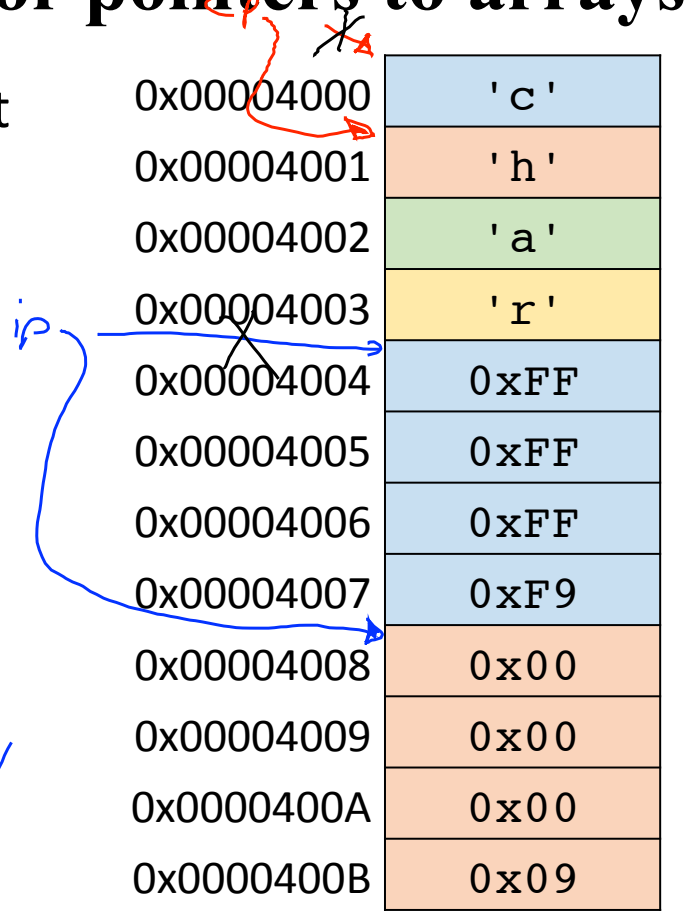
# Pointer arithmetic is useful for pointers to arrays

- 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++;

int *ip = array;
ip++;



# Convert the C code to MIPS assembly to understand what is going on

```
int strlen(char *string) {
 int len = 0;

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


 return len;
}
```

$a0

$v0

```
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
```

# i>clicker

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

```
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!

- http://c-faq.com/decl/spiral.anderson.html

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

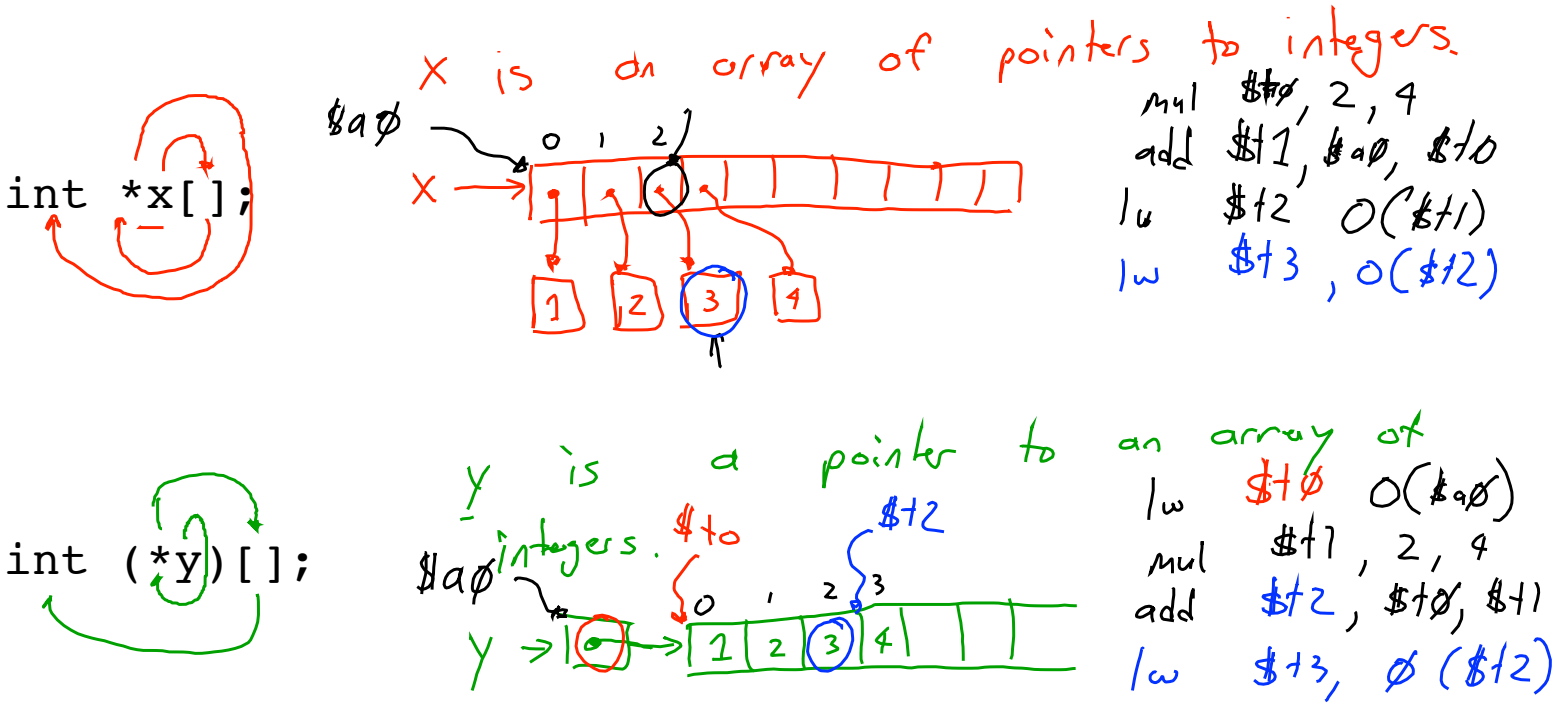
1. [X] or [] => Array X size of... or Array undefined size of...
2. (type1, type2) => function passing type1 and type2 returning...
3. * => pointer(s) to...

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

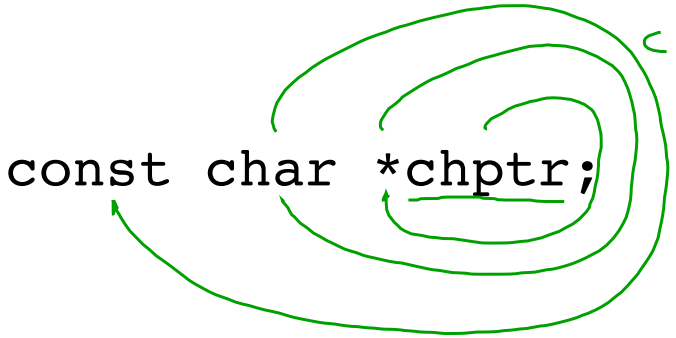
Always resolve anything in parenthesis first!

```
char *str[10];
```

# More Examples (Arrays and Pointers)



# More Examples (Const and Pointers)

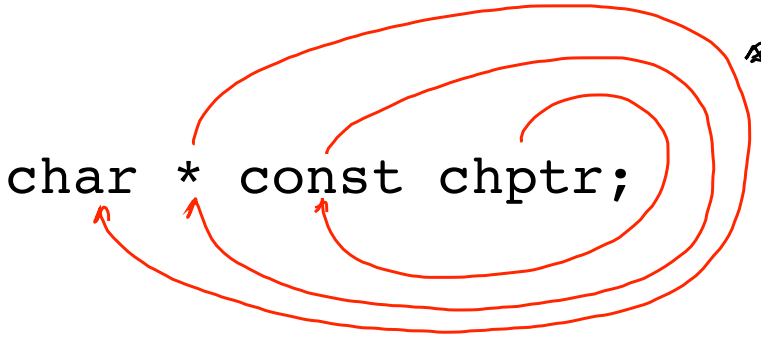



chptr is ptr to a character constant

const char c2 = 'y';

*chptr = 'x'; ✗

chptr = &c2; ✓ (const char *)




chptr = &c2 ✗

*chptr = 'y'; ✓

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)

```
int *z(int);
```

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

```
la $t0, main
jalr $t0
```

```
int f(int);
```

```
int (*q)(int);
```

```
q = &f;
*q(7);
```

```
int *(*w)(int (*x)(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

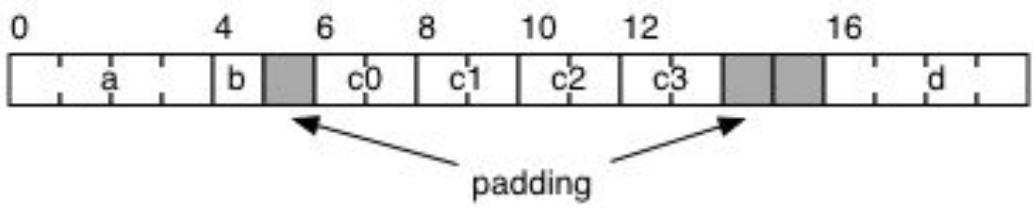
# Compilers/assemblers insert padding to "naturally align" data in structs

- 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

char a;
int b;
char c;
int d;
char e;

≥ 4 B

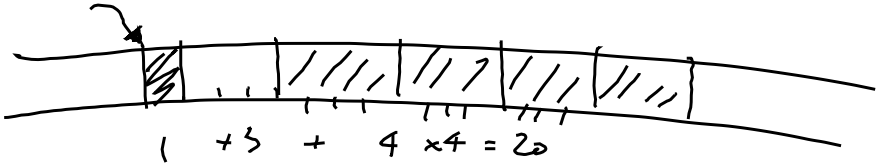
```
struct {
  int a;
  char b;
  short c[4];
  int d;
}
```




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)