CS107, Lecture 10 Extra Practice
Arrays and Pointers, Take II

Reading: K&R (5.2-5.5) or Essential C section 6
Ed Discussion: https://edstem.org/us/courses/28214/discussion/1959584
1. char* vs char[] exercises

Suppose we use a variable `str` as follows:

```c
// initialize as below
A str = str + 1;
B str[1] = 'u';
C printf("%s", str)
```

For each of the following initializations:
- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[7];
   strcpy(str, "Hello1");`
2. `char *str = "Hello2";`
3. `char arr[7];
   strcpy(arr, "Hello3");
   char *str = arr;`
4. `char *ptr = "Hello4";
   char *str = ptr;`
### 1. char* vs char[] exercises

Suppose we use a variable `str` as follows:

```c
// initialize as below
A  str = str + 1;
B  str[1] = 'u';
C  printf("%s", str)
```

For each of the following initializations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[7];
   strcpy(str, "Hello1");`
   
   Line A: Compile error
   (cannot reassign array)

2. `char *str = "Hello2";`
   
   Line B: Segmentation fault
   (string literal)

3. `char arr[7];
   strcpy(arr, "Hello3");
   char *str = arr;`
   
   Prints `eulo3`

4. `char *ptr = "Hello4";
   char *str = ptr;`
   
   Line B: Segmentation fault
   (string literal)
2. Bonus: Tricky addresses

```c
void tricky_addresses() {
    char buf[] = "Local";
    char *ptr1 = buf;
    char **double_ptr = &ptr1;
    printf("ptr1's value: %p\n", ptr1);
    printf("ptr1's deref : %c\n", *ptr1);
    printf("address: %p\n", &ptr1);
    printf("double_ptr value: %p\n", double_ptr);
    printf("buf's address: %p\n", &buf);

    char *ptr2 = &buf;
    printf("ptr2's value: %s\n", ptr2);
}
```

What is stored in each variable?
2. Bonus: Tricky addresses

```c
void tricky_addresses() {
    char buf[] = "Local";
    char *ptr1 = buf;
    char **double_ptr = &ptr1;
    printf("ptr1's value: %p\n", ptr1);
    printf("ptr1's deref: %c\n", *ptr1);
    printf("address: %p\n", &ptr1);
    printf("double_ptr value: %p\n", double_ptr);
    printf("buf's address: %p\n", &buf);

    // Line 10 is a compiler warning
    char *ptr2 = &buf;
    printf("ptr2's value: %s\n", ptr2);
}
```

While Line 10 raises a compiler warning, functionally it will still work—because pointers are addresses.
Translating C into English

* If declaration: “pointer”
  ex: int * is "pointer to an int"
* If operation: "dereference/the value at address"
  ex: *num is "the value at address num"

```
int arr[] = {3, 4, -1, 2}; // initializes stack array
// with 4 ints
1. int *ptr0 = arr;
2. int *elt0 = *arr;
3. int elt = *(arr + 3);
4. int **ptr1 = &ptr0;
```

& "address of"

<table>
<thead>
<tr>
<th>ptr name</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr name</td>
<td>address (except sizeof)</td>
</tr>
</tbody>
</table>
Translating C into English

If declaration: “pointer”
   ex: int * is "pointer to an int"

If operation: "dereference/the value at address"
   ex: *num is "the value at address num"

1. int *ptr0 = arr;
2. int *elt0 = *arr;
3. int elt = *(arr + 3);
4. int **ptr1 = &ptr0;

// initializes stack array
// with 4 ints

Address arr
Value at address arr
The value at address <3 ints after address arr>
address of ptr

Type check with a diagram!
Pen and paper: A * Wars Story

```c
void binky() {
    int a = 10;
    int b = 20;
    int *p = &a;
    int *q = &b;
    *p = *q;
    p = q;
}
```

• Lines 2-5: Draw a diagram.
• Line 7: Update your diagram.
• Line 8: Update your diagram.
void binky() {
    int a = 10;
    int b = 20;
    int *p = &a;
    int *q = &b;
    *p = *q;
    p = q;
}
void binky() {
    int a = 10;
    int b = 20;
    int *p = &a;
    int *q = &b;
    *p = *q;
    p = q;
}
In variable **declaration**, * creates a **pointer**.

```c
char ch = 'r';          // ch stores a char
char *cptr = &ch;       // cptr stores an address of a char (points to a char)
char **strptr = &cptr;  // strptr stores an address of a char * (points to a char *)
```
In **reading values from/storing values**, *dereferences* a pointer.

```c
char ch = 'r';
ch = ch + 1;

char *cptr = &ch;
char **strptr = &cptr;
```

Increment value stored in `ch`
In **reading values from/storing values**, *dereferences* a pointer.

```c
char ch = 'r';
ch = ch + 1;

char *cptr = &ch;
*cptr = *cptr + 1;

char **strptr = &cptr;
```

- Increment value stored in `ch`
- Increment value stored at memory address in `cptr` (increment char pointed to)
In reading values from/storing values, * dereferences a pointer.

```c
char ch = 'r';
ch = ch + 1;

char *cptr = &ch;
*cptr = *cptr + 1;

char **strptr = &cptr;
*strptr = *strptr + 1;
```

Increment value stored in `ch`

Increment value stored at memory address in `cptr` (increment char pointed to)

Increment value stored at memory address in `cptr` (increment address pointed to)
Exercise: Implementation

The below function sums up the string lengths of the num strings in strs.

• Try both 1. array[] syntax and 2. pointer arithmetic!

```c
size_t get_total_strlen(char *strs[], size_t num) {
    size_t total_length = 0;
    for (int i = 0; i < num; i++) {
        // fill this in
    }
    return total_length;
}
```
Exercise: Implementation

The below function sums up the string lengths of the num strings in strs.

• Try both 1. array[] syntax and 2. pointer arithmetic!

```c
size_t get_total_strlen(char *strs[], size_t num) {
    size_t total_length = 0;
    for (int i = 0; i < num; i++) {
        // TODO: fill this in two ways
    }
    return total_length;
}
```

Equivalent:

1. `total_length += strlen(strs[i]);`
2. `total_length += strlen(*(strs + i));`
```c
void skip_spaces(char **p_str) {
    int num = strspn(*p_str, " ");
    *p_str = *p_str + num;
}

int main(int argc, char *argv[]){
    char *str = " Hi!";
    skip_spaces(&str);
    printf("%s", str); // "Hi!"
    return 0;
}
```

What diagram most accurately depicts program state at Line 4 (before `skip_spaces` returns to `main`)?

A. ![Diagram A]
B. ![Diagram B]
C. ![Diagram C]
What diagram most accurately depicts program state at Line 4 (before `skip_spaces` returns to `main`)?