CS107, Lecture 9
Arrays and Pointers

Reading: K&R (5.2-5.5) or Essential C section 6
Ed Discussion: https://edstem.org/us/courses/46162/discussion/3625002
Pointers Summary

• If you are performing an operation with some input and do not care about any changes to the input, **pass the data type itself**.

• If you are modifying a specific instance of some value, **pass the location** of what you would like to modify.

• If a function takes an address (pointer) as a parameter, it can **go to** that address if it needs the actual value.

• If a function accepts an `int *`, it can modify the `int` at the supplied address.

• If a function accepts a `char *`, it can modify the `char` at the supplied address.

• If a function accepts an `char **`, it can modify the `char *` at the supplied address.
Exercise 1

We want to write a function that flips the case of a letter. What should go in each of the blanks?

```c
void flipCase(__?__) {
    if (isupper(__?__)) {
        __?__ = __?__;
    } else if (islower(__?__)) {
        __?__ = __?__;
    }
}

int main(int argc, char *argv[]) {
    char ch = 'g';
    flipCase(__?__);
    printf("%c", ch);  // want this to print ‘G’
}
Exercise 1

We want to write a function that flips the case of a letter. What should go in each of the blanks?

```c
void flipCase(char *letter) {
    if (isupper(*letter)) {
        *letter = tolower(*letter);
    } else if (islower(*letter)) {
        *letter = toupper(*letter);
    }
}

int main(int argc, char *argv[]) {
    char ch = 'g';
    flipCase(&ch);
    printf("%c", ch); // want this to print ‘G’
}
```

We are modifying a specific instance of the letter, so we pass the location of the letter we would like to modify.
Sometimes, we would like to modify a string’s pointer itself, rather than just the characters it points to, e.g., we want to write a function \texttt{skipSpaces} that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

\begin{verbatim}
void skipSpaces(__?__,__) {
    ...
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skipSpaces(__?__,__);
    printf("%s", str); // should print "hello"
}
\end{verbatim}
Exercise 2

Sometimes, we would like to modify a string’s pointer itself, rather than just the characters it points to, e.g., we want to write a function `skipSpaces` that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```c
void skipSpaces(char **strPtr) {
    // code that advances *strPtr
}

int main(int argc, char *argv[]) {
    char *str = "    hello"
    skipSpaces(&str);
    printf("%s", str);  // should print "hello"
}
```

We are modifying a specific instance of the string pointer, so we pass the *location* of the string pointer we would like to modify.
Sometimes, we would like to modify a string’s pointer itself, rather than just the characters it points to, e.g., we want to write a function `skipSpaces` that modifies a string pointer to skip past any initial spaces. What should go in each of the blanks?

```c
void skipSpaces(char *strPtr) {
    // code incapable of modifying str of main
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skipSpaces(str);
    printf("%s", str); // should print "hello", but won't
}
```

This can only advance `skipSpaces`'s own copy of the string pointer, not the instance in main.
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = " hi";
    skipSpaces(&myStr);
    printf("%s
", myStr); // hi
    return 0;
}
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = " hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);    // hi
    return 0;
}
Pointers to Strings

void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = " hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);    // hi
    return 0;
}
Pointers to Strings

```c
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = " hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);    // hi
    return 0;
}
```
Pointers to Strings

```c
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = " hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);    // hi
    return 0;
}
```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = " hi";
    skipSpaces(&myStr);
    printf("%s
", myStr);    // hi
    return 0;
}
Pointers to Strings

```c
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = " hi";
    skipSpaces(&myStr);
    printf("%s \n", myStr);  // hi
    return 0;
}
```
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = " hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);          // hi
    return 0;
}
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *myStr = " hi";
    skipSpaces(&myStr);
    printf("%s\n", myStr);   // hi
    return 0;
}

Weird thought – 0x11 is a string.
Strings In Memory

1. If we create a string as a `char[]`, we can modify its characters because its memory lives in our stack space.

2. We cannot set a `char[]` equal to another value, because it is not a pointer, as it refers to the block of memory reserved for the original array.

3. If we pass a `char[]` as a parameter, set something equal to it, or perform arithmetic with it, it’s automatically converted to a `char *`.

4. If we create a new string with new characters as a `char *`, we cannot modify its characters because its memory lives in the data segment.

5. We can set a `char *` equal to another value, because it is an assignable pointer.

6. Adding an offset to a C string gives us a substring that’s many places past the first character.

7. If we change characters in a string parameter, these changes will persist outside of the function.
String Behavior #1: If we create a string as a char[], we can modify its characters because its memory lives in our stack space.
Character Arrays

When we declare an array of characters, contiguous memory is allocated on the stack to store the contents of the entire array. We can modify what is on the stack.

```c
char str[6];
strcpy(str, "apple");
```
String Behavior #2: We cannot set a `char[]` equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.
An array variable refers to an entire block of memory. We cannot reassign an existing array to be equal to a new array.

```c
char str[6];
strcpy(str, "apple");
char str2[8];
strcpy(str2, "apple 2");

str = str2; // not allowed!
```

An array’s size cannot be changed once we create it; we must create another new array instead.
String Behavior #3: If we pass a char[] as a parameter, set something equal to it, or perform arithmetic with it, it’s automatically converted to a char *.
String Parameters

How do you think the parameter str is being represented?

A. A copy of the array local_str
B. A pointer containing an address to the first element in local_str
String Parameters

How do you think the parameter `str` is being represented?

A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`
How do you think the local variable `str` is being represented?

```c
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = local_str;
    ...
    return 0;
}
```

A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`
How do you think the local variable `str` is being represented?

```c
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = local_str;
    ...
    return 0;
}
```

A. A copy of the array `local_str`

B. A pointer containing an address to the first element in `local_str`
How do you think the local variable `str` is being represented?

A. A copy of part of the array `local_str`
B. A pointer containing an address to the third element in `local_str`
How do you think the local variable `str` is being represented?

```c
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = local_str + 2;
    ...
    return 0;
}
```

A. A copy of part of the array `local_str`

B. A pointer containing an address to the third element in `local_str`
String Parameters

All string functions take char * parameters – they accept char[], but they are implicitly converted to char * before being passed.

- strlen(char *str)
- strcmp(char *str1, char *str2)
- ...

• char * is still a string in all the core ways a char[] is
  • Access/modify characters using bracket notation
  • Print it out
  • Use string functions
  • But under the hood they are represented differently!

• Takeaway: We create strings as char[], pass them around as char *
String Behavior #4: If we create a new string with new characters as a `char *`, we cannot modify its characters because its memory lives in the data segment.
There is another convenient way to create a string if we do not need to modify it later. We can create a `char *` and set it directly equal to a string literal.

```c
char *myString = "Hello, world!";
char *empty = ""

myString[0] = 'h'; // crashes!
printf("%s", myString); // Hello, world!
```
When we declare a char pointer equal to a string literal, the characters are *not* stored on the stack. Instead, they are stored in a special area of memory called the “data segment”. We *cannot modify memory in this segment*.

```c
char *str = "hi";
```

The pointer variable (e.g. `str`) refers to the *address of the first character of the string in the data segment*.

This applies only to creating *new* strings with char *. This does *not* apply for making a char * that points to an existing stack string.
Memory Locations

For each code snippet below, can we modify the characters in \texttt{myStr}?

\begin{verbatim}
char myStr[6];
\end{verbatim}

\textbf{Key Question:} where do its characters live? Do they live in memory we own? Or the read-only data segment?
Memory Locations

For each code snippet below, can we modify the characters in myStr?

```c
char *myStr = "Hi";
```

**Key Question:** where do its characters live? Do they live in memory we own? Or the read-only data segment?
For each code snippet below, can we modify the characters in `myStr`?

```c
char buf[6];
strcpy(buf, "Hi");
char *myStr = buf;
```

**Key Question:** where do its characters live? Do they live in memory we own? Or the read-only data segment?
For each code snippet below, can we modify the characters in `myStr`?

```c
char *otherStr = "Hi";  
char *myStr = otherStr;
```

**Key Question:** where do its characters live? Do they live in memory we own? Or the read-only data segment?
Memory Locations

For each code snippet below, can we modify the characters in `myStr`?

```c
void myFunc(char *myStr) {
    ...
}
```

```c
int main(int argc, char *argv[]) {
    char buf[6];
    strcpy(buf, "Hi");
    myFunc(buf);
    return 0;
}
```

**Key Question:** where do its characters live? Do they live in memory we own? Or the read-only data segment?
Memory Locations

Q: Is there a way to check in code whether a string’s characters are modifiable?
A: No. This is something you can only tell by looking at the code itself and how the string was created.

Q: So then if I am writing a string function that modifies a string, how can I tell if the string passed in is modifiable?
A: You can’t! This is something you instead state as an assumption in your function documentation. If someone calls your function with a read-only string, it will crash, but that’s not your function’s fault :-)

String Behavior #5: We can set a char * equal to another value, because it is an assignable pointer.
A `char *` variable refers to a single character. We can reassign an existing `char *` pointer to be equal to another `char *` pointer.

```c
char *str = "apple"; // e.g. 0xffff0
char *str2 = "apple 2"; // e.g. 0xfee0
str = str2; // ok! Both store address 0xfee0
```
Arrays and Pointers

We can also make a pointer equal to an array; it will point to the first element in that array.

```c
int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    char *ptr = str;
    ...
}
```
Arrays and Pointers

We can also make a pointer equal to an array; it will point to the first element in that array.

```c
int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    char *ptr = str;

    // equivalent
    char *ptr = &str[0];

    // confusingly equivalent, avoid
    char *ptr = &str;
    ...
}
```
String Behavior #6: Adding an offset to a C string gives us a substring that’s many places past the first character.
When we do pointer arithmetic, we are adjusting the pointer by a certain number of places (e.g. characters).

```
char *str = "apple";        // e.g. 0xff0
char *str2 = str + 1;       // e.g. 0xff1
char *str3 = str + 3;       // e.g. 0xff3

printf("%s", str);         // apple
printf("%s", str2);         // pple
printf("%s", str3);         // le
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xff5</td>
<td>'\0'</td>
</tr>
<tr>
<td>0xff4</td>
<td>'e'</td>
</tr>
<tr>
<td>0xff3</td>
<td>'l'</td>
</tr>
<tr>
<td>0xff2</td>
<td>'p'</td>
</tr>
<tr>
<td>0xff1</td>
<td>'p'</td>
</tr>
<tr>
<td>0xff0</td>
<td>'a'</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
When we use bracket notation with a pointer, we are performing \textit{pointer arithmetic and dereferencing}:

```c
char *str = "apple"; // e.g. \texttt{0xff0}
```

// both of these add three places to \texttt{str},
// and then dereference to get the char there.
// E.g. get memory at \texttt{0xff3}.
```
char thirdLetter = str[3];    // 'l'
char thirdLetter = *(str + 3); // 'l'
```
String Behavior #7: If we change characters in a string parameter, these changes will persist outside of the function.
When we pass a `char *` string as a parameter, C makes a *copy* of the address stored in the `char *` and passes it to the function. This means they both refer to the same memory location.

```c
void myFunc(char *myStr) {
    ...
}

int main(int argc, char *argv[]) {
    char *str = "apple";
    myFunc(str);
    ...
}
```
Strings as Parameters

When we pass a **char array** as a parameter, C makes a *copy of the address of the first array element* and passes it (as a **char**) to the function.

```c
void myFunc(char *myStr) {
    ...
}

int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    myFunc(str);
    ...
}
```
Strings as Parameters

When we pass a **char array** as a parameter, C makes a *copy of the address of the first array element* and passes it (as a **char * */) to the function.

```c
void myFunc(char *myStr) {
    ...
}

int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    // equivalent
    char *strAlt = str;
    myFunc(strAlt);
    ...
}
```

### Address Value

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>'a'</td>
</tr>
<tr>
<td>0x101</td>
<td>'p'</td>
</tr>
<tr>
<td>0x102</td>
<td>'p'</td>
</tr>
<tr>
<td>0x103</td>
<td>'l'</td>
</tr>
<tr>
<td>0x104</td>
<td>'e'</td>
</tr>
<tr>
<td>0x105</td>
<td>'\0'</td>
</tr>
</tbody>
</table>

**STACK**

- myStr: 0xf
- main(): ...
- myFunc(): ...

This means if we modify characters in `myFunc`, the changes will persist back in `main`!

```c
void myFunc(char *myStr) {
    myStr[4] = 'y';
}
int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    myFunc(str);
    printf("%s\n", str); // apply ...
}
```
Strings as Parameters

This means if we modify characters in `myFunc`, the changes will persist back in `main`!

```c
void myFunc(char *myStr) {
    myStr[4] = 'y';
}

int main(int argc, char *argv[]) {
    char str[6];
    strcpy(str, "apple");
    myFunc(str);
    printf("%s", str); // apply ...
}
```
Strings In Memory

1. If we create a string as a `char[]`, we can modify its characters because its memory lives in our stack space.

2. We cannot set a `char[]` equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.

3. If we pass a `char[]` as a parameter, set something equal to it, or perform arithmetic with it, it’s automatically converted to a `char *`.

4. If we create a new string with new characters as a `char *`, we cannot modify its characters because its memory lives in the data segment.

5. We can set a `char *` equal to another value, because it is a reassign-able pointer.

6. Adding an offset to a C string gives us a substring that many places past the first character.

7. If we change characters in a string parameter, these changes will persist outside of the function.
Arrays vs. Pointers

• When you create an array, you are making space for each element in the array.
• When you create a pointer, you are making space for a 64-bit address.
• Arrays "decay to pointers" when passed as parameters.
• &arr does nothing on arrays, but &ptr on pointers gets its address
• sizeof(arr) gets the size of an array in bytes, but sizeof(ptr) is always 8