Good news, everyone!

• No class on Wednesday (Oct 28)
• Lab 3 – C-Strings and Valgrind postponed to next week
• Lab 5 – Shell Tools and Scripting canceled
• Assignment 2 will be out on Oct 28 (due Nov 11)
• No office hour this week (Republic Day)
Recap: Pointers

• A *pointer* is a variable that stores a memory address.

• Because there is no pass-by-reference in C like in C++, pointers let us pass around the address of one instance of memory, instead of making many copies.

• One (8 byte) pointer can represent any size memory location!

• Pointers are also essential for allocating memory on the heap, which we will cover later.

• Pointers also let us refer to memory generically, which we will cover later.
Recap: Pointers

• 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.
Recap: Pointers

• **Tip:** setting a function parameter equal to a new value usually doesn’t do what you want. Remember that this is setting the function’s *own copy* of the parameter equal to some new value.

```c
void doubleNum(int x) {
    x = x * x;    // modifies doubleNum’s own copy!
}

void advanceStr(char *str) {
    str += 2;    // modifies advanceStr’s own copy!
}
```
COMP201 Topic 4: How can we effectively manage all types of memory in our programs?
Plan for Today

- Pointers and Parameters (cont’d.)
- Double Pointers
- Arrays in Memory
- Arrays of Pointers

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
Lecture Plan

• Pointers and Parameters (cont’d.)
• Double Pointers
• Arrays in Memory
• Arrays of Pointers
C Parameters

When you pass a value as a parameter, C passes a copy of that value.

```c
void myFunction(int x) {
    ...
}

int main(int argc, char *argv[]) {
    int num = 4;
    myFunction(num);                // passes copy of 4
}
```
C Parameters

When you pass a value as a parameter, C passes a copy of that value.

```c
void myFunction(int *x) {
    ...
}

int main(int argc, char *argv[]) {
    int num = 4;
    myFunction(&num); // passes copy of e.g. 0xffed63
}```
C Parameters

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

```c
void myFunction(char ch) {
    printf("%c", ch);
}

int main(int argc, char *argv[]) {
    char *myStr = "Hello!";
    myFunction(myStr[1]);    // prints 'e'
}
```
C Parameters

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

Do I care about modifying this instance of my data? If so, I need to pass where that instance lives, as a parameter, so it can be modified.
char *

- A char * is technically a pointer to a single character.
- We commonly use char * as string by having the character it points to be followed by more characters and ultimately a null terminator.
- A char * could also just point to a single character (not a string).
Pointers

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

```c
void capitalize(char *ch) {
    // modifies what is at the address stored in ch
}

int main(int argc, char *argv[]) {
    char letter = 'h';
    /* We don’t want to capitalize any instance of 'h'. */
    /* We want to capitalize *this* instance of 'h'! */
    capitalize(&letter);
    printf("%c", letter); // want to print 'H';
}
```
Pointers

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

```c
void doubleNum(int *x) {
    // modifies what is at the address stored in x
}

int main(int argc, char *argv[]) {
    int num = 2;
    /* We don’t want to double any instance of 2. * We want to double *this* instance of 2! */
    doubleNum(&num);
    printf("%d", num); // want to print 4;
}
```
Pointers

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

```c
void capitalize(char *ch) {
    // *ch gets the character stored at address ch.
    char newChar = toupper(*ch);

    // *ch = goes to address ch and puts newChar there.
    *ch = newChar;
}
```
Pointers

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

```c
void capitalize(char *ch) {
    /* go to address ch and put the capitalized version * of what is at address ch there. */
    *ch = toupper(*ch);
}
```
Pointers

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

```c
void capitalize(char *ch) {
    // this capitalizes the address ch! 😞
    char newChar = toupper(ch);

    // this stores newChar in ch as an address! 😞
    ch = newChar;
}
```
Exercise 1

We want to write a function that prints out the square of a number. What should go in each of the blanks?

```c
void printSquare(__?__) {
    int square = __?__ * __?__;
    printf("%d", square);
}

int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(__?__);     // should print 9
}
```
Exercise 1

We want to write a function that prints out the square of a number. What should go in each of the blanks?

```c
void printSquare(int x) {
    int square = x * x;
    printf("%d", square);
}

int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(num);  // should print 9
}
```

We are performing a calculation with some input and do not care about any changes to the input, so we pass the data type itself.
Exercise 1

We want to write a function that prints out the square of a number. What should go in each of the blanks?

```c
void printSquare(int x) {
    x = x * x;
    printf("%d", x);
}

int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(num); // should print 9
}
```

We are performing a calculation with some input and do not care about any changes to the input, so we pass the data type itself.
Exercise 2

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 c) {
    if (isupper(c)) {
        c = tolower(c);
    } else if (islower(c)) {
        c = toupper(c);
    }
}

int main(int argc, char *argv[]) {
    char ch = 'g';
    flipCase(ch);
    printf("%c", ch);  // want this to print ‘G’
}
```
Exercise 2

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.
Lecture Plan

• Pointers and Parameters (cont’d.)
• Double Pointers
• Arrays in Memory
• Arrays of Pointers
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(__?__) {
    ...
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skipSpaces(__?__);
    printf("%s", str);    // should print "hello"
}
```
Exercise 3

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) {
    ...
}

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.
Exercise 3

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) {
    ...
}

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

This advances `skipSpace`'s own copy of the string pointer, not the instance in `main`. 
Demo: Skip Spaces

skip_spaces.c
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;
}
```

```
Address  Value
---------
0x105  0xf
0x13  \0
0x12  i
0x11  h
0x10  
0xf  
...  ...
```
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
", 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\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;
}
```

STACK

DATA SEGMENT

```
Address     Value
0x0      ...
0x105     myStr
0x11      ...
0x13      ' \0'
0x12      'i'
0x11      'h'
0x10      ...
0xf       ...
```
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 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. 

B. 

C. 

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

A. 

B. 

C. 

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

A. 

B. 

C. 

- A.
- B.
- C.
Lecture Plan

• Pointers and Parameters (cont’d.)
• Double Pointers
• Arrays in Memory
• Arrays of Pointers
Arrays

When you declare an array, contiguous memory is allocated on the stack to store the contents of the entire array.

```
char str[6];
strcpy(str, "apple");
```

The array variable (e.g. `str`) is not a pointer; it refers to the entire array contents. In fact, `sizeof` returns the size of the entire array!

```
int arrayBytes = sizeof(str); // 6
```
Arrays

An array variable refers to an entire block of memory. You cannot reassign an existing array to be equal to a new array.

```c
int nums[] = {1, 2, 3};
int nums2[] = {4, 5, 6, 7};
nums = nums2;  // not allowed!
```

An array's size cannot be changed once you create it; you must create another new array instead.
Arrays as Parameters

When you pass an array as a parameter, C makes a copy of the address of the first array element, and passes it (a pointer) to the function.

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

int main(int argc, char *argv[]) {
    char str[3];
    strcpy(str, "hi");
    myFunc(str);
    ...
}
```
Arrays as Parameters

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

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

int main(int argc, char *argv[]) {
    char str[3];
    strcpy(str, "hi");
    // equivalent
    char *arrPtr = str;
    myFunc(arrPtr);
    ...
}
```

**Address/Value Table**:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f2</td>
<td>\0</td>
</tr>
<tr>
<td>0x1f1</td>
<td>i</td>
</tr>
<tr>
<td>0x1f0</td>
<td>h</td>
</tr>
<tr>
<td>0x1e8</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
</tbody>
</table>

**Diagram**:

- `main()` stack frame with `str`, `arrPtr`, and `myStr`.
- `myFunc()` stack frame with pointers and values on the stack.
Arrays as Parameters

This also means we can no longer get the full size of the array using `sizeof`, because now it is just a pointer.

```c
void myFunc(char *myStr) {
    int size = sizeof(myStr); // 8
}

int main(int argc, char *argv[]) {
    char str[3];
    strcpy(str, "hi");
    int size = sizeof(str); // 3
    myFunc(str);
    ...
}
```
`sizeof` returns the size of an array, or 8 for a pointer. Therefore, when we pass an array as a parameter, we can no longer use `sizeof` to get its full size.
Arrays and Pointers

You 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[3];
    strcpy(str, "hi");
    char *ptr = str;
    ...
}
```
Arrays and Pointers

You 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[3];
    strcpy(str, "hi");
    char *ptr = str;

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

    // equivalent, but avoid
    char *ptr = &str;
    ...
}
```
Lecture Plan

• Pointers and Parameters (cont’d.)
• Double Pointers
• Arrays in Memory
• Arrays of Pointers
Arrays Of Pointers

You can make an array of pointers to e.g. group multiple strings together:

```c
char *stringArray[5];  // space to store 5 char *
```

This stores 5 `char *s, not all of the characters for 5 strings!

```c
char *str0 = stringArray[0];  // first char *
```
Arrays Of Pointers

`./swapwords apple banana orange peach pear`
**Arrays Of Pointers**

```
./swapwords apple banana orange peach pear
```

What is the value of `argv[2]` in this diagram?

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x128</td>
<td>0xf8a5 p e a r \0</td>
</tr>
<tr>
<td>0x120</td>
<td>0xf89f p e a c h \0</td>
</tr>
<tr>
<td>0x118</td>
<td>0xf898 o r a n g e \0</td>
</tr>
<tr>
<td>0x110</td>
<td>0xf887 b a n a n a \0</td>
</tr>
<tr>
<td>0x108</td>
<td>0xf881 a p p l e \0</td>
</tr>
<tr>
<td>0x100</td>
<td>0xf838 s w a p w o r d s \0</td>
</tr>
</tbody>
</table>

```
argv = 0x100
argc = 6
```

```
0x128     0xf8a5
0x120     0xf89f
0x118     0xf898
0x110     0xf887
0x108     0xf881
0x100     0xf838
```

.. image:: image.png
Recap

• Pointers and Parameters
• Double Pointers
• Arrays in Memory
• Arrays of Pointers

Next Time: pointer arithmetic, dynamically allocated memory