```c
struct Stack {
    int *stack[stack_size];
    int curIndex;
    int initialized;
};

int StackInit(struct Stack *stack) {
    int i;
    stack->initialized = true;
    for (i = 0; i < stack_size; i++) {
        stack->stack[i] = 0;
    }
    return SUCCESS;
}

int StackGet(struct Stack *stack, int index) {
    if (index >= stack->curIndex) {
        return 0;
    } else {
        return stack->stack[index];
    }
}
```
```c
in1) Slap-Teary (and slat slap slap, char recall, if multi.)

char str[m];
int i;
Sprintf (str, \"slap[ox \x]\n\", slash);
printf (\"multi, str\n\n\"
);
for (i = 0; i < slat.m & current_word[i] != \n; i++) {
    Sprintf (str, \"\%\f\", \%i, i, \f
    i: 5.6
    2: 2.4
}
```
Pointers and Functions

Pointers and Functions

Passing Pointers to Functions

- Normally, functions operate on copies of the data passed to them (pass by value)

```c
int x = 2, y = 0;
int square(int n)
{
    return (n * n);
}
int main(void)
{
    y = square(x);
}
```

Value of variable passed to function is copied into local variable `n`

After Function Call:
- `y = 4`
- `x = 2`

`x` was not changed by function
Pointers and Functions

Passing Pointers to Functions

- Pointers allow a function to operate on the original variable (pass by reference)

```c
int x = 2, y = 0;

void square(int *n)
{
    *n *= *n;
}

int main(void)
{
    square(&x);
}
```

After Function Call: x = 4
x was changed by function

Pointers and Functions

Passing Pointers to Functions

- A function with a pointer parameter:

```
int foo(int *q)
```

- Must be called in one of two ways:
  (assume: `int x, *p = &x;`)

```
foo(&x)  // Pass an address to the function so the address may be assigned to the pointer parameter:
    q = &x
```

```
foo(p)  // Pass a pointer to the function so the address may be assigned to the pointer parameter:
    q = p
```
Pointers and Functions
Passing Parameters By Reference

Example – Part 1
Swap function definition:
```c
void swap(int *n1, int *n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

Addresses of parameters copied to local pointer variables: Function can now modify the original variables via pointers.

We know where you live!

Example – Part 2
Main function definition:
```c
int main(void)
{
    int x = 5, y = 10;
    int *p = &y;
    swap(&x, p);
    while(1);
}
```

Swap function prototype:
```c
void swap(int *n1, int *n2)
```

Tell function where x and y live...
```c
n1 = &x
n2 = p
```

After running program:
```c
x = 10
y = 5
```
Pointers and Strings

• So far, we have worked with strings strictly as arrays of char
• Strings may be created and used with pointers much more elegantly

Pointers and Strings

• When initialized, a pointer to a string points to the first character:

\[ \text{char \ } *\text{str} = \text{"Microchip";} \]

\[ \text{str} \]

\[ \text{str} += 4 \]

• Increment or add an offset to the pointer to access subsequent characters
**Pointers and Strings**

- Pointers may also be used to access characters via an offset:

```c
char *str = "Microchip";
*str == 'M'
*(str + 4) == 'o'
```

- Pointer always points to "base address"

Offsets used to access subsequent chars

**Pointers and Strings**

**Pointer versus Array: Initialization at Declaration**

- Initializing a character string when it is declared is essentially the same for both a pointer and an array:

  **Example: Pointer Variable**

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

  **Example: Array Variable**

  ```c
  char str[] = "PIC";
  ```

  ```c
  char str[4] = "PIC";
  ```

  The NULL character \0 is automatically appended to strings in both cases (array must be large enough).
Pointers and Strings

Pointer versus Array: Assignment in Code

- An entire string may be assigned to a pointer
- A character array must be assigned character by character

Example: Pointer Variable
```c
char *str;
str = "PIC";
```

Example: Array Variable
```c
char str[4];
str[0] = 'P';
str[1] = 'I';
str[2] = 'C';
str[3] = '\0';
```

Must explicitly add NULL character '\0' to array.

Pointers and Strings

Comparing Strings

- If you want to test a string for equivalence, the natural thing to do is:
  ```c
  if (str == "Microchip")
  ```
  - This is **not** correct, though it might appear to work sometimes
  - This compares the address in `str` to the address of the string literal "Microchip"
  - The correct way is to use the `strcmp()` function in the standard library which compares strings character by character

Gabriel Hugh Elkaim – Spring 2013

CMPE-013/L: "C" Programming
Points and Strings
Comparing Strings

- **strcmp()** prototype:

  ```
  int strcmp(const char *s1, const char *s2);
  ```

- **strcmp()** return values:
  - <0 if s1 is less than s2
  - 0 if s1 is equal to s2
  - >0 if s1 is greater than s2

Example:

```c
#include <string.h>

char *str = "Microchip";

int main(void)
{
    if (0 == strcmp(str, "Microchip"))
        printf("They match!\n");

    while(1);
}
```
Arrays of Pointers

Declaration

• An array of pointers is an ordinary array variable whose elements happen to all be pointers.

```c
char *p[4];
```

• This creates an array of 4 pointers to `char`
  – The array `p[]` itself is like any other array
  – The elements of `p[]`, such as `p[1]`, are pointers to `char`

Arrays of Pointers

Array Elements are Pointers Themselves

16-bit Data Memory (RAM)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p[0]</code></td>
<td>0000</td>
<td>91C0</td>
<td>91C3</td>
</tr>
<tr>
<td><code>p[1]</code></td>
<td>91C7</td>
<td>91CC</td>
<td></td>
</tr>
<tr>
<td><code>p[2]</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>p[3]</code></td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

- `p[0]` points to `On` "On"
- `p[1]` points to `Off" Off"
- `p[2]` points to `Main" Main"
- `p[3]` points to `Aux" Aux"
Arrays of Pointers

Initialization

• A pointer array element may be initialized just like its ordinary variable counterpart:

\[
p[0] = &x;
\]

• Or, when working with strings:

\[
p[0] = "My string";
\]

Arrays of Pointers

Dereferencing

• To use the value pointed to by a pointer array element, just dereference it like you would an ordinary variable:

\[
y = *p[0];
\]

• Using \(*p[0]\) is the same as using the object it points to, such as \(x\) or the string literal "My String" from the previous slide.
Arrays of Pointers

Accessing Strings

Example

```c
int i = 0;
char *str[] = {"Zero", "One", "Two", "Three", "Four", "\0"};

int main(void)
{
    while(*str[i] != '\0')
        printf("%s\n", str[i++]);

    while(1);
}
```

Lab Exercise 12

Pointers, Arrays, and Functions
Lab 12
Pointers, Arrays, and Functions

• Open the lab Project:

On the class website
/Examples/Lab12.zip -> Load “Lab12.X”

Exercise 12
Pointers, Arrays, and Functions

Solution: Steps 1 and 2

/*############################################################################
# STEP1: Pass the variable x to the function twosComplement such that the
# value of x itself may be changed by the function. Note: The function
# expects a pointer (address) as its parameter.
########################################################################*/
//Perform twos complement on x
twosComplement(&x);

/*############################################################################
# STEP 2: Pass the array 'a' to the function reverse1(). Use the constant
# ARRAY_SIZE for the second parameter.
# See definition of function reverse1() below.
########################################################################*/
//Reverse order of elements by passing array
reverse1(a, ARRAY_SIZE);
Exercise 12
Pointers, Arrays, and Functions

Solution: Steps 3 and 4

```c
#include <stdio.h>

void reverse2(char *a, size_t ARRAY_SIZE)
{
    // Reverse order of elements by passing pointer
    char *temp;
    for (int i = 0; i < ARRAY_SIZE / 2; i++)
    {
        temp = a + i;
        a[i] = a[ARRAY_SIZE - i - 1];
        a[ARRAY_SIZE - i - 1] = *temp;
    }
}

void twosComplement(int *number)
{
    *number = ~(*number);         // Bitwise complement value
    *number += 1;                 // Add 1 to result
}
```

Exercise 12
Conclusions

- Pointers make it possible to pass a variable by reference to a function (allows function to modify original variable – not a copy of its contents)
- Arrays are frequently treated like pointers
- An array name alone represents the address of the first element of the array
Questions?