Pointers and Functions

Gabriel Hugh Elkaim
Winter 2014

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);
}
```

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);
    y = square(x);
}
```

After Function Call:
- \( y = 4 \)
- \( x = 4 \)
- \( x \) was changed by function

Pointers and Functions

Passing Pointers to Functions

- A function with a pointer parameter:

```c
int foo(int *q)

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;
```

Example

```c
int foo(int *q)
```
### Pointers and Functions

#### Example – Part 1

**Swap function definition:**

```c
void swap(int *n1, int *n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

We know where you live!

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

#### 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...

### 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

**String declaration with a pointer:**

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

Implementation varies depending on compiler and architecture used.

**16-bit Data Memory (RAM):**

<table>
<thead>
<tr>
<th>Address</th>
<th>0x08C2 0x91C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>91C0</td>
</tr>
<tr>
<td>IP</td>
<td></td>
</tr>
<tr>
<td>\0 C</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>49 50 00 43</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x91C0 0x91C2</td>
</tr>
</tbody>
</table>

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

```c
char *str = "Microchip";
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:

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

- Pointer always points to "base address"
- Offsets used to access subsequent chars

Pointers and Strings

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

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

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

Pointers and Strings

- Comparing Strings

  - If you want to test a string for equivalence, the natural thing to do is:
  ```
  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

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

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

```
char str[4];
str[0] = 'P';
str[1] = 'I';
str[2] = 'C';
str[3] = '\0';
```
Pointers 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
Comparing Strings

```
#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.

```
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

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.

Lab Exercise 12

Pointers, Arrays, and Functions

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 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

Exercise 12
Pointers, Arrays, and Functions

Solution: Steps 3 and 4

Exercise 12
Pointers, Arrays, and Functions

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?
# CMPE-013/L: "C" Programming

### Bit 2

- **Set a Bit**: Force bit to 1
- **Clear a Bit**: Force bit to 0
- **Test a Bit**: Return bit from value

```
define Bit2 0x09
```

- **Test Bits**
  - **To Test**: `if` `(v & Bit2)`
    - `Bit2 & v == v`
    - `3 &= 1`
    - `v ^= Bit2`
    - `v = v & Bit2`
  - `v >>= 1`

---

### Bitwise Logic

<table>
<thead>
<tr>
<th>Bitwise Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;</code></td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>^</code></td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td><code>~</code></td>
<td>One's Complement</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>Right Shift</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td>Left Shift</td>
</tr>
</tbody>
</table>

---

### Example

```
v = 0b1010
Bit2 = 0b1000
v &= Bit2  // v becomes 0b1000
v ^= Bit2  // v becomes 0b0000
v >>= 1    // v becomes 0b0000
```