CMPE-013/L

Pointers

Gabriel Hugh Elkaim
Winter 2014

Pointers

Pointers

What are pointers?

A pointer is a variable or constant that holds the address of another variable or function

P = &x;

Pointers

What do they do?

A pointer allows us to indirectly access a variable (just like indirect addressing in assembly language)

p = &x;

x = 0x0123;

*p = 0x0123;

p points to x

Pointers

A Variable’s Address versus A Variable’s Value

In some situations, we will want to work with a variable’s address in memory, rather than the value it contains...

Variable stored at Address

0x0802

0x0804 0x0806 0x0808

0x0800

Variable name from C code

int x;

Value of variable x = 0x0123

Address

16-bit Data Memory (RAM)

05A

0123

DEAD

BEEF

FOOD

0456

x

p

0x0802

0x0804

0x0806

0x0808

0x0800

0x080A

0x0800

0x0802

0x0804

0x0806

0x0808

0x0800

0x080A
**Pointers**

Why would I want to do that?

- Pointers make it possible to write a very short loop that performs the same task on a range of memory locations / variables.

```
//Point to RAM buffer starting address
char *bufPtr = &buffer;

while((DataAvailable) && (ReceivedCharacter != '\0'))
{
    //Read byte from UART and write it to RAM buffer
    ReadUART(bufPtr);
    //Point to next available byte in RAM buffer
    bufPtr++;
}
```

**Pointers**

Where else are they used?

- Used in conjunction with dynamic memory allocation (creating variables at runtime)
- Provide method to pass arguments by reference to functions
- Provide method to pass more than one piece of information into and out of a function
- A more efficient means of accessing arrays and dealing with strings

```
// Create a pointer to int
int *iPtr;

// Create a pointer to float
float *fPtr;
```
Pointers

How to Create a Pointer Type with typedef

typedef type *typeName;

• A pointer variable can now be declared as type typeName which is a synonym for type
• The * is no longer needed since typeName explicitly identifies the variable as a pointer to type

Example

typedef int *intPtr; // Create pointer to int type
intPtr p; // Create pointer to int
          // Equivalent to: int *p;

Pointers

Initialization

• To set a pointer to point to another variable, we use the & operator (address of), and the pointer variable is used without the dereference operator *:

\[ p = \&x; \]

• This assigns the address of the variable \( x \) to the pointer \( p \) (\( p \) now points to \( x \))
• Note: \( p \) must be declared to point to the type of \( x \) (e.g. int \( x; \) int *p;)

Pointers

Usage

• When accessing the variable pointed to by a pointer, we use the pointer with the dereference operator *:

\[ y = *p; \]

• This assigns to the variable \( y \), the value of what \( p \) is pointing to (\( x \) from the last slide)
• Using \(*p\), is the same as using the variable it points to (e.g. \( x \))

Pointers

Another Way To Look At The Syntax

Example

\[ int x, *p; //int and a pointer to int \]
\[ p = \&x; //Assign p the address of x \]
\[ *p = 5; //Same as x = 5; \]

• \&x is a constant pointer
  – It represents the address of \( x \)
  – The address of \( x \) will never change
• \( p \) is a variable pointer to int
  – It can be assigned the address of any int
  – It may be assigned a new address any time
**Pointers**

Another Way To Look At The Syntax

```c
int x, *p;  //1 int, 1 pointer to int
p = &x;    //Assign p the address of x
*p = 5;    //Same as x = 5;
```

- `*p` represents the data pointed to by `p`
  - `*p` may be used anywhere you would use `x`
  - It is the *dereference* operator, also called the *indirection operator*
  - In the pointer declaration, the only significance of `*` is to indicate that the variable is a pointer rather than an ordinary variable

**Pointers**

How Pointers Work

```
{ int x, y;
 int *p;
 x = 0xDEAD;
 y = 0xBEEF;
 p = &x;
 *p = 0x1000;
 p = &y;
 *p = 0x2000;
}
```

**Pointers**

How Pointers Work

```
{ int x, y;
 int *p;
 x = 0xDEAD;
 y = 0xBEEF;
 p = &x;
 p = &y;
 *p = 0x2000;
}```
Example

Pointers
How Pointers Work

{ int x, y;
  int *p;
  x = 0xDEAD;
  y = 0xBEEF;
  p = &x;
  *p = 0x0100;
  p = &y;
  *p = 0x0200;
}

{ int x, y;
  int *p;
  x = 0xDEAD;
  y = 0xBEEF;
  p = &x;
  *p = 0x0100;
  p = &y;
  *p = 0x0200;
}
**Pointers**

How Pointers Work

```c
{  
    int x, y;
    int *p;
    x = 0xDEAD;
    y = 0xBEEF;
    p = &x;
    *p = 0x0100;
    p = &y;
    *p = 0x0200;
}
```

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08BA</td>
<td>0000</td>
</tr>
<tr>
<td>0x08BC</td>
<td>0100</td>
</tr>
<tr>
<td>0x08BE</td>
<td>0200</td>
</tr>
<tr>
<td>0x08C0</td>
<td>0B8E</td>
</tr>
<tr>
<td>0x08C2</td>
<td>0000</td>
</tr>
<tr>
<td>0x08C4</td>
<td>0000</td>
</tr>
<tr>
<td>0x08C6</td>
<td>0000</td>
</tr>
<tr>
<td>0x08C8</td>
<td>0000</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08BA</td>
<td>0000</td>
</tr>
<tr>
<td>0x08BE</td>
<td>0200</td>
</tr>
<tr>
<td>0x08C0</td>
<td>0000</td>
</tr>
<tr>
<td>0x08C2</td>
<td>0000</td>
</tr>
<tr>
<td>0x08C4</td>
<td>0000</td>
</tr>
<tr>
<td>0x08C6</td>
<td>0000</td>
</tr>
<tr>
<td>0x08C8</td>
<td>0000</td>
</tr>
</tbody>
</table>

---

**Pointers and Arrays**

A Quick Reminder...

- Array elements occupy consecutive memory locations

```c
int x[3] = {1, 2, 3};
```

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFF</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0002</td>
<td>0003</td>
</tr>
</tbody>
</table>

---

**Pointers and Arrays**

Initializing a Pointer to an Array

- The array name is the same thing as the address of its first (0th) element

If we declare the following array and pointer variable:

```c
int x[5] = {1, 2, 3, 4, 5};
int *p;
```

We can initialize the pointer to point to the array using any one of these three methods:

- `p = x;` \(\text{Works only for arrays!}\)
- `p = &x;` \(\text{Works for arrays or variables}\)
- `p = &x[0];` \(\text{This one is the most obvious}\)

---

**Pointers and Arrays**

A Preview of Pointer Arithmetic

- Incrementing a pointer will move it to the next element of the array

```c
int x[3] = {1, 2, 3};
int *p;
```

```c
p = &x;  //Works for arrays or variables
p++;  //This one is the most obvious
```

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFF</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0002</td>
<td>0003</td>
</tr>
</tbody>
</table>

More on this in just a bit...
**Pointers and Arrays**

A Preview of Pointer Arithmetic

- Incrementing a pointer will move it to the next element of the array

```
int x[3] = {1, 2, 3};
int *p;
p = &x;
p++;
```

16-bit Data Memory (RAM)

Address

0x0001 0x0800
0x0002 0x0802
0x0003 0x0804
FFFF 0x0806

More on this in just a bit...

**Pointers and Arrays**

A Preview of Pointer Arithmetic

- Incrementing a pointer will move it to the next element of the array

```
int x[3] = {1, 2, 3};
int *p;
p = &x;
p++;
```

16-bit Data Memory (RAM)

Address

0x0001 0x07FE
0x0002 0x0800
0x0003 0x0802
FFFF 0x0806

More on this in just a bit...

**Pointers and Arrays**

A Preview of Pointer Arithmetic

- Incrementing or decrementing a pointer will add or subtract a multiple of the number of bytes of its type

- If we have:

```
float a[4];
float *ptr = &a;
ptr++;
```

16-bit Data Memory Words

Incrementing `ptr` moves it to the next sequential `float` array element

```
float x;
float *p = &x;
p++;
```

We will get `p = &x` since a `float` variable occupies 4 bytes of memory
**Pointer Arithmetic**

**Larger Jumps**

- Adding or subtracting any other number with the pointer will change it by a multiple of the number of bytes of its type
- If we have

```c
int x;
int *p = &x;
p += 3;
```

We will get $p = x + 6$ since an `int` variable occupies 2 bytes of memory.

**Example**

```c
float a[4];
float a[5];
float a[6];
float a[7];
float a[8];
```

```markdown
<table>
<thead>
<tr>
<th>Address</th>
<th>0x0800</th>
<th>0x0802</th>
<th>0x0804</th>
<th>0x0806</th>
<th>0x0808</th>
<th>0x080A</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0001</td>
<td>0002</td>
<td>0003</td>
<td>0000</td>
<td>0002</td>
<td>0000</td>
</tr>
<tr>
<td>x[1]</td>
<td>0000</td>
<td>0002</td>
<td>0003</td>
<td>0000</td>
<td>0002</td>
<td>0000</td>
</tr>
<tr>
<td>x[2]</td>
<td>0000</td>
<td>0002</td>
<td>0003</td>
<td>0000</td>
<td>0002</td>
<td>0000</td>
</tr>
</tbody>
</table>
```

**Example**

```c
long *p = &x;
```

Adding 6 to `ptr` moves it 6 float array elements ahead (24 bytes ahead)

```c
ptr += 6;
```

**Example**

```c
long x[3] = {1, 2, 3};
long *p = &x;
```

```markdown
<table>
<thead>
<tr>
<th>Address</th>
<th>0x0800</th>
<th>0x0802</th>
<th>0x0804</th>
<th>0x0806</th>
<th>0x0808</th>
<th>0x080A</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0001</td>
<td>0002</td>
<td>0003</td>
<td>0000</td>
<td>0002</td>
<td>0000</td>
</tr>
<tr>
<td>x[1]</td>
<td>0000</td>
<td>0002</td>
<td>0003</td>
<td>0000</td>
<td>0002</td>
<td>0000</td>
</tr>
<tr>
<td>x[2]</td>
<td>0000</td>
<td>0002</td>
<td>0003</td>
<td>0000</td>
<td>0002</td>
<td>0000</td>
</tr>
</tbody>
</table>
```

**Example**

```c
long *p = &x;
```

Adding 4 to `p` moves it 4 long elements ahead (32 bytes ahead)

```c
p += 4;
```

**Example**

```c
long x[3] = {1, 2, 3};
long *p = &x;
```

```markdown
<table>
<thead>
<tr>
<th>Address</th>
<th>0x0800</th>
<th>0x0802</th>
<th>0x0804</th>
<th>0x0806</th>
<th>0x0808</th>
<th>0x080A</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0001</td>
<td>0002</td>
<td>0003</td>
<td>0000</td>
<td>0002</td>
<td>0000</td>
</tr>
<tr>
<td>x[1]</td>
<td>0000</td>
<td>0002</td>
<td>0003</td>
<td>0000</td>
<td>0002</td>
<td>0000</td>
</tr>
<tr>
<td>x[2]</td>
<td>0000</td>
<td>0002</td>
<td>0003</td>
<td>0000</td>
<td>0002</td>
<td>0000</td>
</tr>
</tbody>
</table>
```

**Example**

```c
long *p = &x;
```

Adding 2 to `p` moves it 2 long elements ahead (16 bytes ahead)

```c
p += 2;
```

**Example**

```c
long x[3] = {1, 2, 3};
long *p = &x;
```

```markdown
<table>
<thead>
<tr>
<th>Address</th>
<th>0x0800</th>
<th>0x0802</th>
<th>0x0804</th>
<th>0x0806</th>
<th>0x0808</th>
<th>0x080A</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0001</td>
<td>0002</td>
<td>0003</td>
<td>0000</td>
<td>0002</td>
<td>0000</td>
</tr>
<tr>
<td>x[1]</td>
<td>0000</td>
<td>0002</td>
<td>0003</td>
<td>0000</td>
<td>0002</td>
<td>0000</td>
</tr>
<tr>
<td>x[2]</td>
<td>0000</td>
<td>0002</td>
<td>0003</td>
<td>0000</td>
<td>0002</td>
<td>0000</td>
</tr>
</tbody>
</table>
```
Pointers

Pointer Arithmetic

Example

```c
long x[3] = {1, 2, 3};
long *p = &x;

*p += 4;
p++;
*p = 0xDEADBEEF;
p++;
*p = 0xF1D0F00D;
p -= 2;
*p = 0xBADF00D1;
```

16-bit Data Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>0000</th>
<th>0005</th>
<th>0000</th>
<th>0002</th>
<th>0003</th>
<th>0000</th>
<th>0804</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>0x0800</td>
<td>0x0802</td>
<td>0x0804</td>
<td>0x0806</td>
<td>0x0808</td>
<td>0x080A</td>
<td>0x080C</td>
</tr>
</tbody>
</table>

Example

```c
long x[3] = {1, 2, 3};
long *p = &x;

*p += 4;
p++;
*p = 0xDEADBEEF;
p++;
*p = 0xF1D0F00D;
p -= 2;
*p = 0xBADF00D1;
```

16-bit Data Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>0000</th>
<th>0005</th>
<th>0000</th>
<th>0002</th>
<th>0003</th>
<th>0000</th>
<th>0804</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>0x0800</td>
<td>0x0802</td>
<td>0x0804</td>
<td>0x0806</td>
<td>0x0808</td>
<td>0x080A</td>
<td>0x080C</td>
</tr>
</tbody>
</table>

Example

```c
long x[3] = {1, 2, 3};
long *p = &x;

*p += 4;
p++;
*p = 0xDEADBEEF;
p++;
*p = 0xF1D0F00D;
p -= 2;
*p = 0xBADF00D1;
```

16-bit Data Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>0000</th>
<th>0005</th>
<th>0000</th>
<th>0002</th>
<th>0003</th>
<th>0000</th>
<th>0804</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>0x0800</td>
<td>0x0802</td>
<td>0x0804</td>
<td>0x0806</td>
<td>0x0808</td>
<td>0x080A</td>
<td>0x080C</td>
</tr>
</tbody>
</table>

Example

```c
long x[3] = {1, 2, 3};
long *p = &x;

*p += 4;
p++;
*p = 0xDEADBEEF;
p++;
*p = 0xF1D0F00D;
p -= 2;
*p = 0xBADF00D1;
```

16-bit Data Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>0000</th>
<th>0005</th>
<th>0000</th>
<th>0002</th>
<th>0003</th>
<th>0000</th>
<th>0804</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>0x0800</td>
<td>0x0802</td>
<td>0x0804</td>
<td>0x0806</td>
<td>0x0808</td>
<td>0x080A</td>
<td>0x080C</td>
</tr>
</tbody>
</table>
Pointers

Pointers

Example

16-bit Data Memory
(RAM)

Address

0x0800

0x0802

0x0804

0x0806

0x0808

0x080A

0x080C

0x07FE

Example

16-bit Data Memory
(RAM)

Address

0x0800

0x0802

0x0804

0x0806

0x0808

0x080A

0x080C

0x07FE

Example

Post-Increment / Decrement Syntax Rule

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operation</th>
<th>Description by Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>*p++</td>
<td>Post-Increment Pointer</td>
<td>z = *(p++); is equivalent to: z = *p; p = p + 1;</td>
</tr>
<tr>
<td>*(p++)</td>
<td>Post-Increment data pointed to by Pointer</td>
<td>z = (*p)++; is equivalent to: z = *p; *p = *p + 1;</td>
</tr>
</tbody>
</table>

Remember: *(p++) is the same as *p++
Example

**Pointers**

**Post-Increment / Decrement Syntax**

```c
{ int x[3] = {1, 2, 3};
  int y;
  int *p = &x;
  y = 5 + *p;    // y is set to 5
  y = 5 + (*p)++;
}                  // y is now set to 5 + 1
```

Remember:

`*(p++)` is the same as `*p++`

```
Remember:
*(p++) is the same as *p++
```
Pointers
Pre-Increment/Decrement Syntax Rule

• Care must be taken with respect to operator precedence when doing pointer arithmetic:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operation</th>
<th>Description by Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>++*p</td>
<td>Pre-Increment</td>
<td>z = *(++p);</td>
</tr>
<tr>
<td></td>
<td>Pointer</td>
<td>is equivalent to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p = p + 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>z = *p;</td>
</tr>
<tr>
<td>++(*p)</td>
<td>Pre-Increment</td>
<td>z = ++(*p);</td>
</tr>
<tr>
<td></td>
<td>data pointed to</td>
<td>is equivalent to:</td>
</tr>
<tr>
<td></td>
<td>by Pointer</td>
<td>*p = *p + 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>z = *p;</td>
</tr>
</tbody>
</table>

Example
Pre-Increment / Decrement Syntax

```
{  int x[3] = {1, 2, 3};  int y;  int *p = &x;  
  y = 5 + *(++p);  
  y = 5 + ++(*p); }
```

Remember:
* (++p) is the same as *++p
**Pointers**

**Pre-Increment / Decrement Syntax**

```c
int x[3] = {1, 2, 3};
int y;
int *p = &x;
y = 5 + *++p;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>0x0000</th>
<th>0x0001</th>
<th>0x0003</th>
<th>0x0003</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0001</td>
<td>0003</td>
<td>0003</td>
<td>0003</td>
</tr>
<tr>
<td>x[1]</td>
<td>0003</td>
<td>0003</td>
<td>0003</td>
<td>0003</td>
</tr>
<tr>
<td>x[2]</td>
<td>0802</td>
<td>0802</td>
<td>0802</td>
<td>0802</td>
</tr>
<tr>
<td>p</td>
<td>0007</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>y</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
</tr>
</tbody>
</table>

**Example**

```c
int x[3] = {1, 2, 3};
int y;
int *p = &x;
y = 5 + *++p;
```

**Pre-Increment / Decrement Summary**

- The parentheses determine what gets incremented/decremented:
  - Modify the pointer itself
    - `*(++p)` or `*++p` and `*(p++)` or `*p++`
  - Modify the value pointed to by the pointer
    - `++(*p)` and `(*p)++`

**Initialization Tip**

- If a pointer isn't initialized to a specific address when it is created, it is a good idea to initialize it as NUL (`pointing to nowhere`)
- This will prevent it from unintentionally corrupting a memory location if it is accidentally used before it is initialized

```c
int *p = NULL;
```
Exercise 11
Pointers and Pointer Arithmetic

Solution: Steps 1, 2 and 3

```c
/*############################################################################ # STEP 1: Initialize the pointer p with the address of the variable x ############################################################################*/
//Point to address of x
p = &x;

/*############################################################################ # STEP 2: Complete the following printf() functions by adding in the appropriate arguments as described in the control string. ############################################################################*/
/*The variable x is located at address 0x%X
The value of x is %d
The pointer p is located at address 0x%X
The value of p is 0x%X
The value pointed to by *p = %d*/

/*############################################################################ # STEP 3: Write the int value 10 to the location p is currently pointing to. ############################################################################*/
*p = 10;

/*############################################################################ # STEP 4: Increment the value that p points to. ############################################################################*/
//Increment array element's value
(*p)++;

/*############################################################################ # STEP 5: Increment the pointer p so that it points to the next item. ############################################################################*/
//Increment pointer to next array element
p++;
```

Solution: Steps 4 and 5
Exercise 11

Conclusions

- Pointers are variables that hold the address of other variables
- Pointers make it possible for the program to change which variable is acted on by a particular line of code
- Incrementing and decrementing pointers will modify the value in multiples of the size of the type they point to

Questions?

Bance ← kunciul b.i.

"Portb" "SFR" "xch"

Laty → LATPORD ← write to Port 1 "Pin 14V"

PortD → read Port D