"RPN"

\[ 9 \ \& \ (\) \]

\[
\begin{array}{c}
0 \\
\text{push}
\end{array}
\]

\[
\begin{array}{c}
\text{pop}
\end{array}
\]

\[ 9 \ \& \ + \ 17 \ - \ 6 \ \&
\]

---

**CMPE-013/L**

**Pointers**

Gabriel Hugh Elkaim

Spring 2013
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...

Pointers
What are pointers?

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

What do they do?

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

```
Direct Access via x
x = 0x0123; → x

Indirect Access via *p
*p = 0x0123; → p
```

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.

```
Example: Data Buffer

//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

Why would I want to do that?

Example: Data Buffer

RAM buffer allocated over a range of addresses (perhaps an array)

Pseudo-code:
1. Point arrow to first address of buffer
2. Write data from UART to location pointed to by arrow
3. Move arrow to point to next address in buffer
4. Repeat until data from UART is 0, or buffer is full (arrow points to last address of buffer)

16-bit Data Memory
(RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08BA</td>
<td>0123</td>
</tr>
<tr>
<td>0x08BC</td>
<td>4567</td>
</tr>
<tr>
<td>0x08BE</td>
<td>89AB</td>
</tr>
<tr>
<td>0x08C0</td>
<td>CDEF</td>
</tr>
<tr>
<td>0x08C2</td>
<td>1357</td>
</tr>
<tr>
<td>0x08C4</td>
<td>9BDF</td>
</tr>
<tr>
<td>0x08C6</td>
<td>0246</td>
</tr>
<tr>
<td>0x08C8</td>
<td>8ACE</td>
</tr>
</tbody>
</table>

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
Pointers
How to Create a Pointer Variable

Syntax

```
type *ptrName;
```

- In the context of a declaration, the * merely indicates that the variable is a pointer
- type is the type of data the pointer may point to
- Pointer usually described as “a pointer to type”

Example

```
int *iPtr;        // Create a pointer to int
float *fPtr;      // Create a pointer to float
void *vPtr;
```

Pointers
How to Create a Pointer Type with typedef

Syntax

```
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

```c
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

```
t 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`
  - `*` 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
Another Way To Look At The Syntax

Contents of Letter (integer literal 5)

| 5 |

Contents of the Mailbox (variable x or *p)

| *p = 5; |

Address of Mailbox (&x)

| 105 |

Bank of Mailboxes (memory locations)

| p = &x; |

Address on Envelope (pointer p)

| 105 |

Pointers
How Pointers Work

Example

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

16-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x08BA</th>
<th>0x08BC</th>
<th>0x08BE</th>
<th>0x08C0</th>
<th>0x08C2</th>
<th>0x08C4</th>
<th>0x08C6</th>
<th>0x08C8</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0000</td>
<td></td>
<td>0000</td>
<td></td>
<td>0000</td>
<td></td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td>0000</td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td>0000</td>
</tr>
</tbody>
</table>

Example:

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

## How Pointers Work

### Example

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

<table>
<thead>
<tr>
<th>Variable at Address</th>
<th>16-bit Data Memory (RAM)</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x08BA</td>
<td>0x000</td>
</tr>
<tr>
<td>x</td>
<td>DEAD</td>
<td>0x08BC</td>
</tr>
<tr>
<td>y</td>
<td>BEEF</td>
<td>0x08BE</td>
</tr>
<tr>
<td>p</td>
<td></td>
<td>0x08C0</td>
</tr>
<tr>
<td>x</td>
<td>0x0100</td>
<td>0x08C2</td>
</tr>
<tr>
<td>y</td>
<td></td>
<td>0x08C4</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>0x08C6</td>
</tr>
<tr>
<td>y</td>
<td></td>
<td>0x08C8</td>
</tr>
</tbody>
</table>

---

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

Example

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

Example

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

A Quick Reminder...

- Array elements occupy consecutive memory locations

- Pointers can provide an alternate method of accessing array elements

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

16-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FE</th>
<th>0x0800</th>
<th>0x0802</th>
<th>0x0804</th>
<th>0x0806</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0001</td>
<td>0002</td>
<td>0003</td>
<td>FFFF</td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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;`  //Works only for arrays!
- `p = x;`   //Works for arrays or variables
- `p = &x[0];`  //This one is the most obvious
Pointsers 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++;
```

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

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

More on this in just a bit...

16-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>x[0]</th>
<th>x[1]</th>
<th>x[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FE</td>
<td>0001</td>
<td>0002</td>
<td>0003</td>
</tr>
<tr>
<td>0x0800</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0802</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0806</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFFF</td>
<td></td>
<td></td>
<td></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

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

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

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

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

Example

```c
float *ptr;
ptr = &a;
ptr++;  // Incrementing ptr moves it to the next sequential float array element
```

16-bit Data Memory Words

```
0x0050 0x0052 0x0054 0x0056 0x0058 0x005A 0x005C 0x005E 0x0060 0x0062 0x0064 0x0066 0x0068 0x006A 0x006C 0x0070 0x0072 0x0074 0x0076
```

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;  // 2 bytes
int *p = &x;
p += 3;
```

We will get `p = &x + 6` since an `int` variable occupies 2 bytes of memory
**Pointer Arithmetic**

**Larger Jumps**

Example:

```c
float *ptr;
ptr = &a;

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

ptr += 6;
```

16-bit Data Memory Words

---

**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 = 0xBDADF00D;
}
```

16-bit Data Memory (RAM)

---

Gabriel Hugh Elkaim – Spring 2013

CMPE-013/L: "C" Programming
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 (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FE</td>
<td>0000</td>
</tr>
<tr>
<td>0x0800</td>
<td>0005</td>
</tr>
<tr>
<td>0x0802</td>
<td>0000</td>
</tr>
<tr>
<td>0x0804</td>
<td>0002</td>
</tr>
<tr>
<td>0x0806</td>
<td>0000</td>
</tr>
<tr>
<td>0x0808</td>
<td>0003</td>
</tr>
<tr>
<td>0x080A</td>
<td>0000</td>
</tr>
<tr>
<td>0x080C</td>
<td>0800</td>
</tr>
</tbody>
</table>

Gabriel Hugh Elkaim – Spring 2013
CMPE-013/L: "C" Programming
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 (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0000</th>
<th>0005</th>
<th>0000</th>
<th>0000</th>
<th>0003</th>
<th>0000</th>
<th>0804</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FE</td>
<td>0x0800</td>
<td>0x0802</td>
<td>0x0804</td>
<td>0x0806</td>
<td>0x0808</td>
<td>0x0810</td>
<td>0x0812</td>
</tr>
</tbody>
</table>

x[0] = 1
x[1] = 2
x[2] = 3
p = 0x0808
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 (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x0000</th>
<th>0x0005</th>
<th>0x0000</th>
<th>0x0004</th>
<th>0x0006</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FE</td>
<td>0x0800</td>
<td>BEEF</td>
<td>DEAD</td>
<td>F00D</td>
<td>F1D0</td>
</tr>
<tr>
<td>0x0808</td>
<td>0x080A</td>
<td>0808</td>
<td>0x07FE</td>
<td>0x080C</td>
<td></td>
</tr>
</tbody>
</table>

Address 0x0800

- x[0]
- x[1]
- x[2]
Pointers

**Pointer Arithmetic**

Example

```c
{  
  long x[3] = {1, 2, 3};  
  long *p = &x;  
  
  *p += 4;  
  x[0] = 10;  
  p++;  
  x[0] = 20;  
  *p = 0xDEADBEEF;  
  p++;  
  *p = 0xF1D0F00D;  
  p -= 2;  
  *p = 0xBADF00D1;  
}
```

16-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0800</td>
<td>0000</td>
</tr>
<tr>
<td>0x0802</td>
<td>00D1</td>
</tr>
<tr>
<td>0x0804</td>
<td>BADF</td>
</tr>
<tr>
<td>0x0806</td>
<td>BEEF</td>
</tr>
<tr>
<td>0x0808</td>
<td>DEAD</td>
</tr>
<tr>
<td>0x080A</td>
<td>F00D</td>
</tr>
<tr>
<td>0x080C</td>
<td>F1D0</td>
</tr>
<tr>
<td>0x080E</td>
<td>0800</td>
</tr>
</tbody>
</table>

Post-Increment/Decrement Syntax Rule

- Care must be taken with respect to operator precedence when doing pointer arithmetic:
  - `(*p)++` is equivalent to: `z = *p; p = p + 1;`
# Pointers

## Post-Increment / Decrement Syntax

**Example**

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

```
<table>
<thead>
<tr>
<th>Address</th>
<th>16-bit Data Memory (RAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FE</td>
<td>0000</td>
</tr>
<tr>
<td>0x0800</td>
<td>0001</td>
</tr>
<tr>
<td>0x0802</td>
<td>0002</td>
</tr>
<tr>
<td>0x0804</td>
<td>0003</td>
</tr>
<tr>
<td>0x0806</td>
<td>0004</td>
</tr>
<tr>
<td>0x0808</td>
<td>0005</td>
</tr>
<tr>
<td>0x080A</td>
<td>0006</td>
</tr>
<tr>
<td>0x080C</td>
<td>0007</td>
</tr>
</tbody>
</table>
```

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

Example

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

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

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

16-bit Data Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>0x7FE</th>
<th>0x800</th>
<th>0x802</th>
<th>0x804</th>
<th>0x806</th>
<th>0x808</th>
<th>0x80A</th>
<th>0x80C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td>0001</td>
<td>0002</td>
<td>0003</td>
<td>0003</td>
<td>0802</td>
<td>0007</td>
<td>0000</td>
</tr>
</tbody>
</table>

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><code>++*p</code></td>
<td>Pre-Increment Pointer</td>
<td><code>z = *++p;</code> is equivalent to: <code>p = p + 1;</code></td>
</tr>
<tr>
<td><code>*(++p)</code></td>
<td>Pre-Increment data pointed to by Pointer</td>
<td><code>z = ++*p;</code> is equivalent to: <code>*p = *p + 1;</code></td>
</tr>
<tr>
<td><code>++(*p)</code></td>
<td>Pre-Increment data pointed to by Pointer</td>
<td><code>z = ++(*p);</code></td>
</tr>
</tbody>
</table>

Remember:

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

Example

```c
{  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 / Decrease Syntax

Example

```c
{  
  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);  
  y = 5 + ++(*p);  
}
```

**Address**

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0800</td>
<td>0001</td>
</tr>
<tr>
<td>0x0802</td>
<td>0003</td>
</tr>
<tr>
<td>0x0804</td>
<td>0003</td>
</tr>
<tr>
<td>0x0806</td>
<td>0802</td>
</tr>
<tr>
<td>0x0808</td>
<td>0008</td>
</tr>
<tr>
<td>0x080A</td>
<td>0000</td>
</tr>
<tr>
<td>0x080C</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Remember:**

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

---

**Pointers**

**Pre- and Post- 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)++
Pointers

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 NULL (pointing to nowhere)
• This will prevent it from unintentionally corrupting a memory location if it is accidentally used before it is initialized

Example

```c
int *p = NULL;
```

NUL is the character ‘\0’ but NULL is the value of a pointer that points to nowhere

Exercise 11

Pointers and Pointer Arithmetic
Lab 11
Pointers and Pointer Arithmetic

- Open the lab Project:

On the class website
/Examples/Lab11.zip -> Load “Lab11.X”

1 Open MPLAB®X and select Open Project Icon (Ctrl + Shift + O)
Open the Project listed above.

⚠️ If you already have a project open in MPLAB X, close it by “right clicking” on the open project and selecting “Close”

Exercise 11
Pointers and Pointer Arithmetic

Solution: Steps 1, 2 and 3

//STEP 1: Initialize the pointer p with the address of the variable x
p = &x;

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

//STEP 3: Write the int value 10 to the location p is currently pointing to.
*p = 10;
Exercise 11

Pointers and Pointer Arithmetic

Solution: Steps 4 and 5

```c
/*############################################################################
# STEP 4: Increment the value that p points to.
############################################################################*/
//Increment array element's value
(*p)++;
printf("y[%d] = %d\n", i, *p);

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

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?