CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Pointers

Pointers and memory
Pointer/array equivalency
Pointer arithmetic
Pointers and the stack
Pointers and strings
Arrays of pointers
Pointers
How to Create a Pointer Variable

Syntax

```c
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

```c
int *iPtr;       // Create a pointer to int
int *iPtr, x;    // Create a pointer to int and an int
float *fPtr1, *fPtr2; // Create 2 float pointers
char *foo;
```
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

Dereferencing

- When accessing the data 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

Dereferencing example

Example

```c
int x = 6, *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 memory value
  - 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

Dereferencing example

Example

```c
int x = 6, *p; // int and a 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 view

Contents of the Mailbox
(variable \( x \))

Address of Mailbox
\((\&x)\)

Bank of Mailboxes
(memory locations)
Pointers

Another view

Contents of the Mailbox

\((x, *p)\)

Address of Mailbox
\((&x, p)\)

Bank of Mailboxes
(memory locations)

\(p = &x;\)
Pointers

Another view

Contents of the Mailbox

\[(x, \*p)\]

Address of Mailbox

\[(\&x, p)\]

*\(p = 2;\)

Bank of Mailboxes
(memory locations)
Pointers
Dereferencing non-primitives

Example

Complex x = \{0.6, 1.2\}, *p;
p = &x;
p->re = 5;

- \textit{p->re} represents the data pointed to by \textit{p}
  - \textit{p->re} may be used anywhere you would use \textit{x.re}
  - \textit{->} is the structure dereference operator, equivalent to
    (*p).re
  - In the pointer declaration, the only significance of * is to
    indicate that the variable is a pointer rather than an ordinary
    variable
void MyFunc(Complex *x)
{
    Complex t = *x;

    x->re /= t.re * t.re + t.im * t.im;
    x->im /= t.re * t.re + t.im * t.im;
}
Pointers
Dereferencing non-primitives

```c
void MyFunc(Complex *x)
{
    Complex t = *x;
}
```
Pointers
Dereferencing non-primitives

```c
void MyFunc(Complex *x)
{
    Complex t = *x;
}
```
Pointers
How Pointers Work

Example

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

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Variable at Address</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0x08B8</td>
</tr>
<tr>
<td>y</td>
<td>0x08BC</td>
</tr>
<tr>
<td>p</td>
<td>0x08C0</td>
</tr>
<tr>
<td></td>
<td>0x08C4</td>
</tr>
<tr>
<td></td>
<td>0x08C8</td>
</tr>
<tr>
<td></td>
<td>0x08CC</td>
</tr>
<tr>
<td></td>
<td>0x08D0</td>
</tr>
<tr>
<td></td>
<td>0x08D4</td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
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

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

Example

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

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08B8</td>
<td>0000</td>
</tr>
<tr>
<td>0x08BC</td>
<td>0000</td>
</tr>
<tr>
<td>0x08C0</td>
<td>0000</td>
</tr>
<tr>
<td>0x08C4</td>
<td>0000</td>
</tr>
<tr>
<td>0x08C8</td>
<td>0000</td>
</tr>
<tr>
<td>0x08CC</td>
<td>0000</td>
</tr>
<tr>
<td>0x08D0</td>
<td>0000</td>
</tr>
<tr>
<td>0x08D4</td>
<td>0000</td>
</tr>
<tr>
<td>x</td>
<td>0000</td>
</tr>
<tr>
<td>y</td>
<td>0000</td>
</tr>
<tr>
<td>p</td>
<td>08BC</td>
</tr>
</tbody>
</table>
Pointers

How Pointers Work

Example

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

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08B8</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08BC</td>
<td>0000 0100</td>
</tr>
<tr>
<td>0x08C0</td>
<td>0000 BEEF</td>
</tr>
<tr>
<td>0x08C4</td>
<td>0000 08BC</td>
</tr>
<tr>
<td>0x08C8</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08CC</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08D0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08D4</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>
Pointers
How Pointers Work

Example

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

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x08B8</th>
<th>0x08BC</th>
<th>0x08C0</th>
<th>0x08C4</th>
<th>0x08C8</th>
<th>0x08CC</th>
<th>0x08D0</th>
<th>0x08D4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0000</td>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0000</td>
<td>BEEF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>0000</td>
<td>08C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Pointers
How Pointers Work

Example

```c
{  
  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

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

![Memory Diagram]

- Pointers can provide an alternate method for accessing array elements
Pointers and Arrays

Initializing a Pointer to an Array

- The array name evaluates to 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 either of these methods:

```c
p = x;  // Works only for arrays
p = &x[0];  // Same as the above
```
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;
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
<th>0x0804</th>
<th>0x0808</th>
<th>0x080C</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0000 0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td>0000 0002</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td>0000 0003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>FFFF FFFF</td>
<td></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;
```

- 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;
p = x + 1;
```

32-bit Data Memory (RAM)

```
<table>
<thead>
<tr>
<th>Address</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td>FFFF</td>
<td>FFFF</td>
</tr>
<tr>
<td>0x0800</td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0x0804</td>
<td>0000</td>
<td>0002</td>
</tr>
<tr>
<td>0x0808</td>
<td>0000</td>
<td>0003</td>
</tr>
<tr>
<td>0x080C</td>
<td>0000</td>
<td>0804</td>
</tr>
</tbody>
</table>
```

- More on this in just a bit...
Pointer Arithmetic

Incrementing Pointers

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

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

We will the address of p incremented by 4 since a float occupies 4 bytes
**Pointer Arithmetic**

**Incrementing Pointers**

**Example**

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

Incrementing `ptr` moves it to the next sequential `float` array element (4 bytes ahead)

32-bit Data Memory Words

Maxwell James Dunne
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
short int x;
short int *p = &x;
p += 3;
```

We will get the address of `p` incremented by 6 since a `short 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)

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

Pointer Arithmetic

Example

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

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 0xBAD00000F001;
}
```
```c
long long x[] = {1, 2, 3};
long long *p = x;

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 0xBAD0000F00D1;
```
Pointers

Pointer Arithmetic

Example

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

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 0.BAD0000F00D1;
```
Pointers

Pointer Arithmetic

Example

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

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 0.BAD0000F00D1;
```

32-bit Data Memory
(RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0000 0000</th>
<th>0000 0005</th>
<th>0000 0000</th>
<th>0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0808</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x080C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0810</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0814</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0818</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

x[0] 0000 0005
x[1] 1234 BEEF
x[2] 0000 DEAD
p 0000 0808
Pointers

Pointer Arithmetic

Example

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

    *p += 4;
    ++p;
    *p = 0xDEAD1234BEEF;
    ++p;
    *p = 0xF1D04321F00D;
    p -= 2;
    *p = 0xBAD0000F00D1;
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
<th>0x0804</th>
<th>0x0808</th>
<th>0x080C</th>
<th>0x0810</th>
<th>0x0814</th>
<th>0x0818</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>x[0]</td>
<td>0000</td>
<td>0005</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td>1234</td>
<td>BEEF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td></td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td>0810</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Maxwell James Dunne

CMPE-013/L: “C” Programming
Pointers

Pointer Arithmetic

Example

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

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 0xBAD0000F00D1;
```
Pointers

Pointer Arithmetic

Example

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

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 0xBAD0000F00D1;
```
Pointers

Pointer Arithmetic

Example

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

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p = 2;
*p = 0xBAD0000F00D1;
```
# Pointers

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

Post-Increment / Decrement Syntax

Example

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

32-bit Data Memory (RAM)

- `x[0]`: 0000 0001
- `x[1]`: 0000 0002
- `x[2]`: 0000 0003

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)++;
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
<th>0x0804</th>
<th>0x0808</th>
<th>0x080C</th>
<th>0x0810</th>
<th>0x0814</th>
<th>0x0818</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>x[0]</td>
<td>001</td>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td>002</td>
<td>0002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td>003</td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td>0800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td>0006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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)++;
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
<th>0x0804</th>
<th>0x0808</th>
<th>0x080C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000 0000</td>
<td>0000 0001</td>
<td>0000 0002</td>
<td>0000 0003</td>
<td>0000 0804</td>
</tr>
</tbody>
</table>

Remember:

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

```
32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0000 0000</th>
<th>0000 0001</th>
<th>0000 0002</th>
<th>0000 0003</th>
<th>0000 0804</th>
<th>0000 0007</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0808</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x080C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0810</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0814</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0818</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Remember: 

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

Post-Increment / Decrement Syntax

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

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0000 0000</th>
<th>0000 0001</th>
<th>0000 0003</th>
<th>0000 0003</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800</td>
<td>0000 0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0808</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x080C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0810</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0814</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0818</td>
<td></td>
<td></td>
<td></td>
<td></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>+++*p</td>
<td>Pre-Increment Pointer</td>
<td>( z = * (++p) ; ) 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 data pointed to by</td>
<td>( z = * * (++) p; ) is equivalent to:</td>
</tr>
<tr>
<td></td>
<td>Pointer</td>
<td>( *p = *p + 1; )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( z = *p; )</td>
</tr>
</tbody>
</table>
Pointers

Pre-Increment / Decrement Syntax

Example

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

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
<th>0x0804</th>
<th>0x0808</th>
<th>0x080C</th>
<th>0x0810</th>
<th>0x0814</th>
<th>0x0818</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0000 0000</td>
<td>0000 0001</td>
<td>0000 0002</td>
<td>0000 0003</td>
<td>0000 0800</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>x[1]</td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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);
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0000 0000</th>
<th>0000 0001</th>
<th>0000 0002</th>
<th>0000 0003</th>
<th>0000 0804</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0808</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x080C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0810</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0814</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0818</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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);
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
<th>0x0804</th>
<th>0x0808</th>
<th>0x080C</th>
<th>0x0810</th>
<th>0x0814</th>
<th>0x0818</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</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);
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0800</td>
<td>0000 0001</td>
</tr>
<tr>
<td>0x0804</td>
<td>0000 0003</td>
</tr>
<tr>
<td>0x0808</td>
<td>0000 0003</td>
</tr>
<tr>
<td>0x080C</td>
<td>0000 0804</td>
</tr>
<tr>
<td>0x0810</td>
<td>0000 0007</td>
</tr>
<tr>
<td>0x0814</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0818</td>
<td>0000 0000</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);
}
```

32-bit Data Memory (RAM)

```
Address       0x07FC  0x0800  0x0804  0x0808  0x080C  0x0810  0x0814  0x0818
0000 0000
0000 0001  0000 0003
0000 0003
0000 0003
0000 0804
0000 0008
0000 0000
0000 0000
```

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

\[
* (++p) \text{ or } +++p \text{ and } *(p++) \text{ or } *p++
\]

Modify the value pointed to by the pointer

\[
++(*p) \text{ and } (*p)++
\]
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 only.
- This will prevent it from unintentionally corrupting a memory location if it is accidentally used before it is initialized.

Example

```
int *p = NULL;  // \0
```

**NUL** is the character '\0' but **NULL** is the value of a pointer that points to nowhere.
int *foo;

int *foo = NULL;

*foo = 3;

~7000 resets per second
Pointers and the Stack

Beware the stack

- Memory addresses may not always be valid
- Addresses referring to the stack have a lifetime tied to that variables scope
- Only global, static, and pointers returned by malloc() will always be valid
- You should almost never use the memory addresses of variables on the stack
Pointers and the Stack

Beware the stack

Example function

```c
int *foo(int x, int y)
{
    int z = x + (++y);
    return &z;
}
```

Example caller

```c
int *main(void)
{
    int a = 6, b = 19;
    int *c = foo(a, b);
    printf("%d\n", *c);
}
```
Pointers and the Stack

Beware the stack

Example function

```c
int *foo(int x, int y)
{
    int z = x + (++y);
    return &z;
}
```

Example caller

```c
int main(void)
{
    int a = 6, b = 19;
    int *c = foo(a, b);
    printf("%d\n", *c);
}
```
Pointers and the Stack

Beware the stack

Example function

```c
int *foo(int x, int y)
{
    int z = x + (++y);
    return &z;
}
```

Example caller

```c
{
    int a = 6, b = 19;
    int *c = foo(a, b);
    puts("Hey!");
    printf("%d\n", *c);
}
```
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 = "PIC32MX"; str
```

Implementation varies depending on compiler and architecture used.
Pointers and Strings

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

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

- Increment or add an offset to the pointer to access subsequent characters:

```c
str += 4
```
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

- Depending on variable type, part of the variable is constant

Example: Pointer to String Constant

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

Example: Character array

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

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

Pointer versus Array: Initialization at Declaration

Example: Pointer Variable

```c
char *str1 = "PIC";
char str2[] = "PIC";
```

<table>
<thead>
<tr>
<th>RAM</th>
<th>ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9D0008C0</td>
<td>0x9D0008C0</td>
</tr>
<tr>
<td>0xA0000FB0</td>
<td>0xA0000FB0</td>
</tr>
<tr>
<td>0xA0000FB4</td>
<td>0xA0000FB4</td>
</tr>
<tr>
<td>0xA0000FB5</td>
<td>0xA0000FB5</td>
</tr>
<tr>
<td>0xA0000FB6</td>
<td>0xA0000FB6</td>
</tr>
<tr>
<td>0xA0000FB7</td>
<td>0xA0000FB7</td>
</tr>
</tbody>
</table>
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

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

Example: Array Variable

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

Must explicitly add NUL character '\0' to array.
Pointers and Strings

Comparing Strings

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

Array Elements are Pointers Themselves

32-bit Data Memory (RAM)

```
0000 0000
9D00 3FC0
9D00 3FC3
9D00 3FC7
9D00 3FCC
0000 0000
0000 0000
0000 0000
0000 0000
```

- 3FC0: On
- 3FC3: Off
- 3FC7: Main
- 3FCC: 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
Different from Two-dimensional Array

```c
char p[4][] = {
    "On",
    "Off",
    "Main",
    "Aux"
};
```

- This creates a two-dimensional array of `chars`
  - Amount of memory for every string the same
# Arrays of Pointers

Array Elements are Sequential

<table>
<thead>
<tr>
<th>32-bit Data Memory (RAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{On \ 0 \ 0 \ 0}</td>
</tr>
<tr>
<td>\texttt{Off \ 0 \ 0 \ 0}</td>
</tr>
<tr>
<td>\texttt{Main \ 0}</td>
</tr>
<tr>
<td>\texttt{Aux \ 0 \ 0 \ 0}</td>
</tr>
</tbody>
</table>

- \texttt{On \ 0 \ 0 \ 0} at \texttt{3FA4-3FA8}
- \texttt{Off \ 0 \ 0 \ 0} at \texttt{3FA9-3FAD}
- \texttt{Main \ 0} at \texttt{3FAE-3FB2}
- \texttt{Aux \ 0 \ 0 \ 0} at \texttt{3FB3-3FB7}

Maxwell James Dunne
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);
}
```
Dynamic Memory

malloc()
free()
Dynamic Memory

Rationale

• Memory needs not known at compile time
• Memory needs to persist outside of current scope
Dynamic Memory

Syntax

```c
void *malloc(size_t size);
```

- Request memory of `size` bytes
  - Usually returned by `sizeof` operator
- Returns valid pointer or `NULL`

Example

```c
typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
```
Dynamic Memory

malloc()'d memory

Example

typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
printf("Complex{re:%f im:%f}\n",
    x->re, x->im);
Dynamic Memory

malloc()'d memory

Example

typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));

x->re = 0.0;

x->im = 0.0;

printf("Complex\{re:%f im:%f\n",
        x->re, x->im);
Dynamic Memory

The Heap

Example

typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
Dynamic Memory

The Heap

Example

typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
Complex *x = malloc(sizeof(Complex));
Complex *x = malloc(sizeof(Complex));
Complex *x = malloc(sizeof(Complex));
Complex *x = malloc(sizeof(Complex));
Complex *x = malloc(sizeof(Complex));

...
Dynamic Memory

NULL pointers

Example

typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
x->re = 0.0;
x->im = 0.0;
printf("Complex{re:%f im:%f}\n", x->re, x->im);
Dynamic Memory
NULL pointers

Example

typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
Complex y = *x;
Dynamic Memory

NULL pointers

Example

typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
if (x) {
    x->re = 0.0;
}

x->im = 0.0;
printf("Complex{re:%f im:%f}\n", x->re, x->im);
Dynamic Memory

NULL pointers

Example

typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
if (x) {
    x->re = 0.0;
    x->im = 0.0;
    printf("Complex{re:%f im:%f}\n", x->re, x->im);
}
Dynamic Memory

`free()`

**Syntax**

```c
void free(void *ptr);
```

- Frees memory pointed to by `ptr`
  - **Must** have been returned by `malloc()`

**Example**

```c
typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
free(x);
```
Dynamic Memory

Invalid free()ing

Example

```c
// Non-initialized pointers
Complex *x;
free(x); // Invalid!

// NULL pointers
Complex *y = NULL;
free(y); // Invalid!

// Non-heap pointers
char *z = "Hey!";
free(z); // Invalid!

// Heap pointers not returned by malloc()
Complex *w = malloc(sizeof(Complex));
free(&w->re); // Invalid!
```
CMPE-013/L

RPN (stack)

Maxwell James Dunne
Stack Operation
struct Stack {
    float stackItems[STACK_SIZE];
    int currentIdx;
    uint8_t initialized;
};

unsigned char
Stack Functions

- `void StackInit(struct Stack *stack)`

  ```c
  struct Stack foo;
  struct Stack *foo;
  StackInit(foo);
  ```

- `int StackPop(struct Stack *stack, float *value);`

  ```c
  return SIZE_ERROR;
  return -1;
  ```
RPN

- Reverse Polish Notation
  - Calculator that uses a stack as a scratchpad
  - We will read strings from the keyboard and parse

- Simple Examples
  - $4 \ 3 \ + = 4 + 3$
  - $1 \ 6 \ -3 \ - \ + = 6 -3 = 9$
  - $2 \ 1 \ \sqrt{} \ 9$
RPN

Error/End Conditions

- The RPN calculation is complete when the whole string has been processed.
- If only one item is left on the stack the string entered was valid.
- It can also fail in a few ways
  - Not one Item left on stack at end
  - Invalid RPN Token
  - Operator without at least 2 items on the stack
  - Stack is full (Specific to the class, not RPN in general)
RPN

Complex Example

• \((1 + 4) \times (6 - 4) / 8\) as \(1 4 + 6 4 - * 8 /\)
Development Tips

'' - 1.25 ''

$q + \text{of}''-''\ 0$

$q + \text{of}''0''\)