CMPE-013/L

Introduction to “C” Programming

Bryant Wenborg Mairs

Spring 2014
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
```
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 *:

\[
\text{int } *x\text{=}\text{ NULL;} \\
x = \&y;
\]

\[
p = \&x;
\]

• 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. \(\text{int } x; \text{ int } *p;\))
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

\begin{verbatim}
int x = 6, *p; // int and a pointer to int

p = &x;       // Assign p the address of x
*p = 5;       // Same as x = 5;
\end{verbatim}

- \(*p\) 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)

p = &x;
Pointers
Another view

Contents of the Mailbox

(x, *p)

Address of Mailbox

(&x, p)

*p = 2;

printf("%d", x);

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

**Example**

```
Complex x = {0.6, 1.2}, *p;
p = &x;
p->re = 5;  // <=> x.re = 5;
```

- `p->re` represents the data pointed to by `p`
  - `p->re` may be used anywhere you would use `x.re`
  - `->` 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
Pointers
Dereferencing non-primitives

Example

```c
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;
}
```

```c
main()
{
    MyFunc();
    // code...
}
```
Pointers
Dereferencing non-primitives

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

```c
void MyFunc(Complex *x)
{
    Complex t = *x;
}
```
### 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>32-bit Data Memory (RAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0000 0000</td>
</tr>
<tr>
<td>y</td>
<td>0000 0000</td>
</tr>
<tr>
<td>p</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>

<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 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>y</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>p</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</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>0000 0000</th>
<th>0000 DEAD</th>
<th>0000 0000</th>
<th>0000 0000</th>
<th>0000 0000</th>
<th>0000 0000</th>
<th>0000 0000</th>
<th>0000 0000</th>
<th>0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08B8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08BC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08D0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08D4</td>
<td></td>
<td></td>
<td></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 = 0x0100;
    p = &x;
    *p = 0x0100;
    p = &y;
    *p = 0x0200;
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08B8</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0x08BC</td>
<td>DEAD</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0x08CC</td>
<td>BEEF</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0x08CD</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0x08DE</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

Variable at Address

- **x**: 0xDEAD
- **y**: 0xBEEF
- **p**: Pointer to **x**
- **p**: Pointer to **y**
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>0x08CC</th>
<th>0x08D0</th>
<th>0x08D4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000 0000</td>
<td>0000 08BC</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>

Variable at Address

- `x`: 0000 0000 0000 DEAD
- `y`: 0000 0000 0000 BEEF

Example of 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>0000</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08B8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08BC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08D0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08D4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### How Pointers Work

Pointers are variables that store the memory address of another variable. They allow us to access and manipulate data stored in memory locations. In the example:

- `int x, y;` declares two integer variables `x` and `y`.
- `int *p;` declares a pointer variable `p` of type `int`.
- `&x` returns the memory address of `x`.
- `x = 0xDEAD;` assigns the value 0xDEAD to `x`.
- `p = &x;` assigns the memory address of `x` to `p`.
- `*p = 0x0100;` dereferences `p` and assigns the value 0x0100 to `x`.
- `p = &y;` assigns the memory address of `y` to `p`.
- `*p = 0x0200;` dereferences `p` and assigns the value 0x0200 to `y`. 
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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td>0100</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>BEEF</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>08C0</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

Variable at Address

- **x**: 0x08B8
- **y**: 0x08C0
- **p**: 0x08D0
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>0200</td>
<td>08C0</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pointers and Arrays
A Quick Reminder...

- Array elements occupy consecutive memory locations

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

- 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}; // x[0] \rightarrow int *
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

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

32-bit Data Memory
(RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>FFFF</th>
<th>FFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td></td>
<td></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>FFFF</td>
<td>FFFF</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[0];
++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</td>
<td>0001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td>0000</td>
<td>0002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td>0000</td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>0000</td>
<td>0800</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- More on this in just a bit...
Pointsers 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 += 1;
```

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
Example

```c
float *ptr;

ptr = &a;

++ptr;
```

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

```
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
**Example**

```c
float *ptr;

ptr = a; // Initialize pointer to start of array

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

ptr += 6;
```

16-bit Data Memory Words

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0050</td>
<td></td>
</tr>
<tr>
<td>0x0054</td>
<td></td>
</tr>
<tr>
<td>0x0058</td>
<td></td>
</tr>
<tr>
<td>0x005C</td>
<td></td>
</tr>
<tr>
<td>0x0060</td>
<td></td>
</tr>
<tr>
<td>0x0064</td>
<td></td>
</tr>
<tr>
<td>0x0068</td>
<td></td>
</tr>
<tr>
<td>0x006C</td>
<td></td>
</tr>
<tr>
<td>0x0070</td>
<td></td>
</tr>
<tr>
<td>0x0074</td>
<td></td>
</tr>
</tbody>
</table>
Example

```c
{  
    long long x[] = {1, 2, 3};
    long long *p = x;
    long long (*p) += 4;
    ++p;
    *p = 0xDEAD1234BEEF;
    ++p;
    *p = 0xF1D04321F00D;
    p -= 2;
    *p = 0xBAD0000F00D1;
}
```
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>0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0005</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800</td>
<td>0x0804</td>
<td>0x0808</td>
<td>0x080C</td>
<td>0x0810</td>
<td>0x0814</td>
<td>0x0818</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CMPE-013/L: “C” Programming

Gabriel Hugh Elkaim – Winter 2014
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>0x0810</th>
<th>0x0814</th>
<th>0x0818</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000 0000</td>
<td>0000 0005</td>
<td>0000 0000</td>
<td>0000 0002</td>
<td>0000 0000</td>
<td>0000 0003</td>
<td>0000 0808</td>
</tr>
</tbody>
</table>
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>x[0]</td>
<td>0000 0000</td>
<td>0000 0005</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td>1234 BEEF</td>
<td>0000 DEAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td>0000 0003</td>
<td>0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>0000 0808</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

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

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

```c
{  
  long long x[] = {1, 2, 3};
  long long *p = x;
  x[0] += 4;
  *p += 4;
  ++p;  // x[1] +=
  *p = 0xDEAD1234BEEF;
  ++p;  // x[2] =
  *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>0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0005</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1234 BEEF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 DEAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 F1D0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0810</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Address 0x07FC: 0000 0000
- Address 0x0800: 0000 0005
- Address 0x0804: 0000 0000
- Address 0x0808: 1234 BEEF
- Address 0x080C: 0000 DEAD
- Address 0x0810: 0000 F1D0
- Address 0x0814: 0000 0810
- Address 0x0818: 0000 0810
Example

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

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
*p = 0xBAD0000F00D1;
```

32-bit Data Memory (RAM)

```
0x0800: 0000 0000
0x0802: 1234 BEEF
0x0804: 0000 DEAD
0x0806: 4321 F00D
0x0808: 0000 F1D0
0x080A: 0000 0800
```

Address

0x07FE
0x0800
0x0802
0x0804
0x0806
0x0808
0x080A
0x080C
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>$z = *(p++)$; 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>
<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>
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)

Address
0x07FC
0x0800
0x0804
0x0808
0x080C
0x0810
0x0814
0x0818

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)++;
}
```

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)++;  
}
```

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 0002</td>
</tr>
<tr>
<td>0x0808</td>
<td>0000 0003</td>
</tr>
<tr>
<td>0x0810</td>
<td>0000 0804</td>
</tr>
<tr>
<td>0x0814</td>
<td>0000 0007</td>
</tr>
<tr>
<td>0x0818</td>
<td>0000 0000</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++
```
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 data pointed to by Pointer</td>
<td>$z = ++(*p);$ is equivalent to: $*p = *p + 1;$</td>
</tr>
<tr>
<td></td>
<td>Pre-Increment Pointer</td>
<td>$z = *(*p);$ is equivalent to: $p = p + 1;$</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></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>
</tbody>
</table>

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);
```

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></td>
<td>0000 0000</td>
<td>0000 0001</td>
<td>0000 0002</td>
<td>0000 0003</td>
<td>0000 0804</td>
<td>0000 0007</td>
<td>0000 0000</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));
}
```

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>0000 0000</td>
<td>0000 0001</td>
<td>0000 0003</td>
<td>0000 0804</td>
<td>0000 0007</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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);
}
```

32-bit Data Memory (RAM)

```
0000 0000
0000 0001
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

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

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

• This will prevent it from unintentionally corrupting a memory location if it is accidentally used before it is initialized

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

NULL is the character ' \0 ' but NULL is the value of a pointer that points to nowhere
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 variable's 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 += 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

• 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";
```

```
RAM
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9D0008C0</td>
<td>0xA0000FB0</td>
</tr>
<tr>
<td>P</td>
<td>0xA0000FB4</td>
</tr>
<tr>
<td>I</td>
<td>0xA0000FB5</td>
</tr>
<tr>
<td>C</td>
<td>0xA0000FB6</td>
</tr>
<tr>
<td>\0</td>
<td>0xA0000FB7</td>
</tr>
</tbody>
</table>

ROM
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>0x9D0008C0</td>
</tr>
<tr>
<td>I</td>
<td>0x9D0008C1</td>
</tr>
<tr>
<td>C</td>
<td>0x9D0008C2</td>
</tr>
<tr>
<td>\0</td>
<td>0x9D0008C3</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

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

Example: Array Variable

```c
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 \text{str} to the address of the string literal \text{"Microchip"}

• The correct way is to use the \text{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.

\[
\text{char } *\text{p}[4];
\]

- This creates an array of 4 pointers to \text{char}
  - The array \text{p[ ]} itself is like any other array
  - The elements of \text{p[ ]}, such as \text{p[1]}, are pointers to \text{char}
Arrays of Pointers

Array Elements are Pointers Themselves

32-bit Data Memory (RAM)

```
p[0] 9D00 3FC0
p[1] 9D00 3FC3
p[2] 9D00 3FC7
p[3] 9D00 3FCC
```

```
0000 0000
0000 0000
0000 0000
0000 0000
```

```
3FC0
3FC3
3FC7
3FCC
```

```
On \0
Off \0
Main \0
Aux \0
```
Arrays of Pointers

Initialization

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

```c
p[0] = &x;
```

• Or, when working with strings:

```c
p[0] = "My string";
```
Arrays of Pointers
Different from Two-dimensional Array

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

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

Array Elements are Sequential

32-bit Data Memory (RAM)

$p[0]$

0000 0000
0000 6E4F
6666 4F00
614D 0000
4100 6E69
0000 7875
0000 0000
0000 0000

$p[1]$

$p[2]$

$p[3]$

3FA4-3FA8

On \0\0\0

3FA9-3FAD

Off \0\0

3FAE-3FB2

Main \0

3FB3-3FB7

Aux \0\0
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.
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

malloc()

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

Syntax

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

Example

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

Complex *x = malloc(sizeof(Complex));
```
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));
```c
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));
```
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;
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);
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);
} else {
    printf("\n");
}
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

// 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!