Definitions:

**Structures** are collections of variables grouped together under a common name. The variables within a structure are referred to as the structure’s **members**, and may be accessed individually as needed.

- **Structures:**
  - May contain any number of members
  - Members may be of **any** data type
  - Allow a group of related variables to be treated as a single unit, even if different types
  - Ease the organization of complicated data
# Structures

## Declaring

**Syntax**

```
struct StructName {
    type_1 memberName_1;
    ...
    type_n memberName_n;
};
```

Members are declared just like ordinary variables.

**Example**

```c
// Structure to handle complex numbers
struct Complex {
    float re;  // Real part
    float im;  // Imaginary part
};
```
Structures
Instantiating

Syntax

```c
struct StructName {
    type_1 memberName_1;
    ...
    type_n memberName_n;
} varName_1, ..., varName_n;
```

Example

```c
// Structure to handle complex numbers
struct Complex {
    float re;
    float im;
} x, y;  // Declare x and y of type complex
```
If `StructName` has already been defined:

```c
struct StructName varName1, ..., varName_n;
```

Example:

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

struct Complex x, y;  // Declare x and y of type complex
```
Structures
Accessing members

Syntax

structVariableName.memberName

Example

```c
struct Complex {
    float re;
    float im;
} x, y;  // Declare x and y of type `struct complex`

int main(void)
{
    x.re = 1.25;  // Initialize real part of x
    x.im = 2.50;  // Initialize imaginary part of x
    y = x;        // Set struct y equal to struct x
    ...
```
Structures
Initialization

Syntax

If `StructName` has already been defined:

```c
struct StructName varName = {const_1, ..., const_n};
```

Example

```c
struct Complex {
    float re;
    float im;
};
...
struct Complex x = {1.25, 2.50};
```
Structures
Nesting Structures

Example

```c
struct point {
    float x;
    float y;
};

struct line {
    struct point a;
    struct point b;
};
```

```c
int main(void)
{
    struct line m = {{{1.2, 7.6}, {38.5, 17.8}}};
    ...
    m.a.x = 0;
}
```
Structures
Nesting Structures

Example

```c
struct point {
    float x;
    float y;
};

struct line {
    struct point a;
    struct point b;
};

int main(void)
{
    struct line m = {{1.2, 7.6}, {38.5, 17.8}};
    printf("Line (%f, %f) <-> (%f, %f)\n",
            m.a.x, m.a.y, m.b.x, m.b.y);
    ...
}
```
Structures
Arrays and Pointers with Strings

- Strings:
  - May be assigned directly to `char` array member only at declaration
  - May be assigned directly to a pointer to `char` member at any time

Example: Structure

```c
struct Strings {
    char a[4];
    char *b;
} str = {"Bad", "Good"};
```

Example: Initializing Members

```c
int main(void)
{
    struct Strings str;
    str.a[0] = 'B';
    str.a[1] = 'a';
    str.a[2] = 'd';
    str.a[3] = '\0';
    str.b = "Good";
}
```
Structures
Creating Arrays of Structures

Syntax
If *StructName* has already been defined:

```c
struct  StructName arrName[n];
```

Example
```c
struct Complex {
    float re;
    float im;
};
...
struct Complex a[3];
```
Structures

Initializing Arrays of Structures at Declaration

**Syntax**

If `StructName` has already been defined:

```c
struct StructName arrName[n] = {{{list_1}}, ..., {{list_n}}};
```

**Example**

```c
struct Complex {
    float re;
    float im;
};
...
struct Complex a[3] = {{{1.2, 2.5}}, {{3.9, 6.5}}, {{7.1, 8.4}}};
```
Structures
Using Arrays of Structures

If `arrName` has already been defined:

**Syntax**

```
arrName[n].memberName
```

**Example: Definitions**

```c
typedef struct {
    float re;
    float im;
} Complex;
...
struct Complex a[3];
```

**Example: Usage**

```c
int main(void)
{
    a[0].re = 1.25;
    a[0].im = 2.50;
    ...
}
```
Structures
Creating a Pointer to a Structure

Syntax

If StructName has already been defined:

```c
struct StructName *ptrName;
```

Example

```c
struct Complex {
    float re;
    float im;
};
...
struct Complex *a;
```
Structures
How to Use a Pointer to Access Structure Members

If $ptrName$ has already been defined:

**Syntax**

```
ptrName->memberNameName
```

Pointer must first be initialized to point to the address of the structure itself: $ptrName = &structVariable$;

**Example: Definitions**

```c
struct Complex {
    float re;
    float im;
};
...
struct Complex x;
struct Complex *p;
```

**Example: Usage**

```c
int main(void)
{
    p = &x;
    // Set x.re = 1.25 via p
    p->re = 1.25;
    // Set x.im = 2.50 via p
    p->im = 2.50;
}
```
Structures
How to Pass Structures to Functions

Example

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

void Display(struct Complex x)
{
    printf("(\%f + j\%f)\n", x.re, x.im);
}

int main(void)
{
    struct Complex a = {1.2, 2.5};
    struct Complex b = {3.7, 4.0};

    Display(a);
    Display(b);
}
```
# Structures

**How to Pass Structures to Functions**

## Example

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

void Display(struct Complex *x)
{
    printf(“(%f + j%f)\n”, x->re, x->im);
}

int main(void)
{
    struct Complex a = {1.2, 2.5};
    struct Complex b = {3.7, 4.0};

    Display(&a);
    Display(&b);
}
```
# Structures

How to Pass Structures to Functions

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

void Display(const struct Complex *x)
{
    printf("(\%f + j\%f)\n", x->re, x->im);
}

int main(void)
{
    struct Complex a = {1.2, 2.5};
    struct Complex b = {3.7, 4.0};

    Display(&a);
    Display(&b);
}
```
typedef
typedef

- Assign new names to existing datatypes
- Interpreted by the compiler (unlike `#define` )

**Syntax**

```
typedef datatype  typeName;
```

- `typedef int Length;`
- `typedef float single;`
- `Length x = 3;`
typedef

How to Create a Structure Type with typedef

**Syntax**

typedef struct StructTag\_optional {
  \_\_type\_\_1 memberName\_\_1;
  ...
  type\_\_n memberName\_\_n;
} TypeName;

**Example**

// Structure type to handle complex numbers
typedef struct {
  float re;     // Real part
  float im;     // Imaginary part
} Complex;

Complex x;
typedef
Declaring structs

Example

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

struct {
    float re;
    float im;
} Complex;

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

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

Maxwell James Dunne
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 \( \text{x} = \{0.6, 1.2\} \), *p;
p = &x;
p->\text{re} = 5;

- \( p->\text{re} \) represents the data pointed to by \( p \)
  - \( p->\text{re} \) may be used anywhere you would use \( x.\text{re} \)
  - \( \rightarrow \) is the structure dereference operator, equivalent to \( (*p).\text{re} \)
  - 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;
}
```
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;
}

32-bit Data Memory (RAM)

Variable at Address

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

Maxwell James Dunne
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

```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>x</th>
<th>y</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08B8</td>
<td>0000 DEAD</td>
<td>0000 BEEF</td>
<td>0000 0200</td>
</tr>
<tr>
<td>0x08BC</td>
<td>0000</td>
<td>0000</td>
<td>0000 0100</td>
</tr>
<tr>
<td>0x08C0</td>
<td>0000</td>
<td>0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08C4</td>
<td>0000</td>
<td>0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08C8</td>
<td>0000</td>
<td>0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08CC</td>
<td>0000</td>
<td>0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08D0</td>
<td>0000</td>
<td>0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08D4</td>
<td>0000</td>
<td>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;
}
```
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;  
}
```

32-bit Data Memory (RAM)

```
Address
0x08B8
0x08BC
0x08C0
0x08C4
0x08C8
0x08CC
0x08D0
0x08D4

Variable at Address

0000 0000
0000 0100
0000 BEEF
0000 08C0
0000 0000
0000 0000
0000 0000
0000 0000
```
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>x</th>
<th>y</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08B8</td>
<td>0000 0100</td>
<td>0000 0200</td>
<td>0000 08C0</td>
</tr>
<tr>
<td>0x08BC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08CC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08D0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08D4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Maxwell James Dunne
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};
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>x[0]</th>
<th>x[1]</th>
<th>x[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td>0000 0001</td>
<td>0000 0002</td>
<td>0000 0003</td>
</tr>
<tr>
<td>0x0800</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0808</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x080C</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;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>32-bit Data Memory (RAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td>FFFF FFFF</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>0x080C</td>
<td>0000 0800</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;
p = x + 1;
```
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)

```
float a[0]
float a[1]
float a[2]
float a[3]
float a[4]
float a[5]
float a[6]
float a[7]
float a[8]
```

```
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 = 0.BAD0000F00D1;
```

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 0000</td>
</tr>
<tr>
<td>0x0808</td>
<td>0000 0002</td>
</tr>
<tr>
<td>0x080C</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0810</td>
<td>0000 0003</td>
</tr>
<tr>
<td>0x0814</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0818</td>
<td>0000 0800</td>
</tr>
</tbody>
</table>
```
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 = 0xBAD0000F00D;
```

32-bit Data Memory (RAM)

```
0000 0000
0000 0005
0000 0000
0000 0002
0000 0000
0000 0003
0000 0000
0000 0800
```
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;
}
```

32-bit Data Memory

<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 0005</td>
</tr>
<tr>
<td>0x0804</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0808</td>
<td>0000 1234</td>
</tr>
<tr>
<td>0x080C</td>
<td>BEEF 0000</td>
</tr>
<tr>
<td>0x0810</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0814</td>
<td>0000 0003</td>
</tr>
<tr>
<td>0x0818</td>
<td>0808 0000</td>
</tr>
</tbody>
</table>

Maxwell James Dunne
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;  
}
```

![Diagram showing memory allocation and pointer arithmetic](image)
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;
```
Example

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

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 0xBA00000F00D1;
```
# 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:</td>
</tr>
<tr>
<td>*(p++)</td>
<td>Pointer</td>
<td>( z = *p ; )</td>
</tr>
<tr>
<td>((p)++)</td>
<td>Post-Increment data pointed to by Pointer</td>
<td>( z = (*p)++ ; ) 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)

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

32-bit Data Memory (RAM)

- Address: 0x07FC
- 0x0800
- 0x0804
- 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++);  // Equivalent to 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>0x080C</td>
<td>0000 0804</td>
</tr>
<tr>
<td>0x0810</td>
<td>0000 0006</td>
</tr>
<tr>
<td>0x0814</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0818</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>

* (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>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

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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000 0000</td>
<td>0000 0001</td>
<td>0000 0003</td>
</tr>
<tr>
<td>x[0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>0000 0804</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0000 0007</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><code>++*p</code></td>
<td>Pre-Increment Pointer</td>
<td><code>z = * (++p);</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>is equivalent to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>p = p + 1;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>z = *p;</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>
<tr>
<td></td>
<td></td>
<td>is equivalent to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>*p = *p + 1;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>z = *p;</code></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>
</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>
</tr>
<tr>
<td>x[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>p</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>y</td>
</tr>
<tr>
<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>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>0000</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</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></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**

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

32-bit Data Memory (RAM)

- `x[0]`: Address 0x07FC, value 0000 0001
- `x[1]`: Address 0x0800, value 0000 0003
- `x[2]`: Address 0x0804, value 0000 0003

- `p`: Address 0x0804, value 0000 0804
- `y`: Address 0x0810, value 0000 0007

- `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 0003</td>
<td>0000 0003</td>
<td>0000 0804</td>
<td>0000 0008</td>
<td>0000 0000</td>
<td>0000 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

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

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
```

`NULL` is the value of a pointer that points to nowhere, not the character `\0`. The `\0` character is the null terminator used in strings.
int *foo;

int *foo = NULL;

*foo = ?;

~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**

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

**Example caller**

```
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:
```
cchar *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
\text{str} = 0
```

- 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";
```

<table>
<thead>
<tr>
<th></th>
<th>RAM</th>
<th>ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>str1</td>
<td>0x9D0008C0</td>
<td>0x9D0008C0</td>
</tr>
<tr>
<td>str2</td>
<td>0xA0000FB0</td>
<td>0xA0000FB0</td>
</tr>
<tr>
<td>P</td>
<td>0xA0000FB4</td>
<td>0x9D0008C0</td>
</tr>
<tr>
<td>I</td>
<td>0xA0000FB5</td>
<td>0x9D0008C1</td>
</tr>
<tr>
<td>C</td>
<td>0xA0000FB6</td>
<td>0x9D0008C2</td>
</tr>
<tr>
<td>\0</td>
<td>0xA0000FB7</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";
```

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.

```
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)

```
  p[0]  9D00 3FC0
     0000 0000
  p[1]  9D00 3FC3
     0000 0000
  p[2]  9D00 3FC7
     0000 0000
  p[3]  9D00 3FCC
     0000 0000
```

- p[0] points to 3FC0
  - On

- p[1] points to 3FC3
  - Off

- p[2] points to 3FC7
  - Main

- p[3] points to 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][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>p[0] 0000 0000</td>
</tr>
<tr>
<td>p[0] 0000 6E4F</td>
</tr>
<tr>
<td>p[1] 6666 4F00</td>
</tr>
<tr>
<td>p[2] 614D 0000</td>
</tr>
<tr>
<td>p[3] 4100 6E69</td>
</tr>
<tr>
<td>0000 7875</td>
</tr>
<tr>
<td>0000 0000</td>
</tr>
<tr>
<td>0000 0000</td>
</tr>
</tbody>
</table>

- p[0] points to `On
  0 0 0 0`
- p[1] points to `Off
  f f 0 0`
- p[2] points to `Main
  i n 0`
- p[3] points to `Aux
  x 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.
```
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()

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));

Heap (top)
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));

Heap (top)

NULL
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;
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

// 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!