CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Spring 2015
Text I/O
Text I/O

• Within `<stdio.h>`:
  – Formatted text: `scanf()`/`printf()`
  – Characters: `getchar()`/`putchar()`
  – Strings/Lines: `fgets()`/`puts()`
  • NEVER EVER EVER USE `gets()`
Text I/O

fgets()

Syntax

```c
char *fgets(char *str, int count, FILE *stream);
```

- `str` is where received data is stored
  - Needs to be an array
- `count` is how many characters to process
  - Stops when \n or `(count-1)` chars are received
- `stream` is stdin
```c
#include <stdio.h>

int main(void)
{
    // Create enough memory for a 50 char string
    char inputData[50 + 1];

    fgets(inputData, sizeof(inputData), stdin);
}
```
String Processing
String Processing

- Within `<string.h>`:
  - Examination
    - Length: `strlen()`
    - Comparing: `strcmp()`/`strncmp()`
    - Splitting: `strtok()`
  - Manipulation
    - Copying: `strncpy()` (Don't use `strcpy()`!)
    - Appending: `strncat()`
String Processing

strlen()

Syntax

```c
size_t strlen(const char *str);
```

- `str` is the string to calculate the length of
- `size_t` can be treated as an `int`

Examples

```c
int x = strlen("My string"); // x = 9

char str[] = "asdf";
int y = strlen(str); // y = 4
```
String Processing

strcmp()

**Syntax**

```
int strcmp(const char *s1, const char *s2);
```

- Ignores size of the strings, purely alphabetical comparison
- Return value is > 0 if s1 alphabetically before s2, 0 if they're equal, < 0 if s2 alphabetically before s1

**Examples**

```c
char *s1 = "apple", *s2 = "zed";
int cmpResult = strcmp(s1, s2);
if (cmpResult > 0) {
    printf("apple > zed\n");
} else if (cmpResult == 0) {
    printf("apple == zed\n" );
} else {
    printf("apple < zed\n");
}
```
String Processing

strtok()

**Syntax**

```c
char *strtok(char *s1, const char *s2);
```

- **s1** (input/output), string to be tokenized
  - Will be modified!
- **s2** (input) – Delimiters

**Examples**

```c
char s1[] = "This is an example!";

char *firstToken = strtok(s1, " "); // firstToken = "This"

char *secondToken = strtok(NULL, " "); // secondToken = "is"

char *thirdToken = strtok(NULL, " "); // thirdToken = "an"

char *fourthToken = strtok(NULL, " "); // fourthToken = "example!"
```
String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
```

s1

```
This is an example! \0
```
## String Processing

### strtok() Details

**Example**

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
```

```
firstToken
s1
```

```
This \0 is an example! \0
```
String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
char *secondToken = strtok(NULL, " ");
```

firstToken
s1

```
This \0 is \0 an example ! \0
```

secondToken
String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
char *secondToken = strtok(NULL, " ");
char *thirdToken = strtok(NULL, " ");
```

```
This 0 is 0 an 0 example ! 0
```

```
firstToken
s1
thirdToken
```

```
secondToken
```
String Processing
stroken() Details

Example

char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
char *secondToken = strtok(NULL, " ");
char *thirdToken = strtok(NULL, " ");
char *fourthToken = strtok(NULL, " ");

firstToken
s1
thirdToken

This \0 is \0 an \0 example \0 ! \0

secondToken

fourthToken
String Processing

strncpy()

Syntax

```c
char *strncpy(char *s1, const char *s2, size_t n);
```

- **s1** (output) – where the string will be copied to
- **s2** (input) - the string that to be copied
- **n** - how many characters can be copied
- Undefined if s1 and s2 overlap!

Examples

```c
char s1[50];
strncpy(s1, "asdf", 4); // s1 = "asdf\0"
strncpy(s1 + strlen(s1), "asdf", 4); // s1 = "asdfasdf\0"
```
String Processing

strncat()

Syntax

```c
char *strncat(char *s1, const char *s2, size_t n);
```

- **s1** (input/output) - is the base string
- **s2** (input) - the string that will be appended
- **n** - how many characters can be appended
- Undefined if s1 and s2 overlap!

Examples

```c
char s1[50] = "This is an example!";
strncat(s1, "asdf", 4);
```
String Processing

- Within `<stdlib.h>`:
  - Conversion
    - Integer: `atoi()`, `xtoi()`
    - Floats: `atof()`
- Within `<stdio.h>`:
  - Conversion
    - Any: `sscanf()`
String Processing

atof()

Syntax

```c
double atof(const char *s);
```

- `s` (input) – The string to parse
- Returns the converted value or 0.0

Examples

```c
char s1[] = "1.03";
double x = atof(s); // y = 1.03

char s2[] = "efg";
double y = atof(s); // y = 0.0
```
Pointers

- Pointers and memory
- Pointer/array equivalency
- Pointer arithmetic
- Pointers and the stack
- Pointers and strings
- Arrays of pointers
Pointers
Address versus value

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

Variable stored at Address

32-bit Data Memory (RAM)

Variable name from C code
int x;

Value of variable
x = 0x0123

Address of variable x = 0x0804
Pointers
What are pointers?

- A pointer holds the address of another variable or function
Pointers

What do they do?

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

Direct Access via \(x\)

\[ x = 0x0123; \rightarrow x \]

Indirect Access via \(*p\)

\[ *p = 0x0123; \rightarrow p \]

\[ p \text{ points to } x. \]
Pointers

Why would I want to do that?

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

Example: Data Buffer

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

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

Why would I want to do that?

Example: Data Buffer

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

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

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08BC</td>
<td>0000 0123</td>
</tr>
<tr>
<td>0x08C0</td>
<td>0000 4567</td>
</tr>
<tr>
<td>0x08C4</td>
<td>0000 89AB</td>
</tr>
<tr>
<td>0x08C8</td>
<td>0000 CDEF</td>
</tr>
<tr>
<td>0x08CC</td>
<td>0000 1357</td>
</tr>
<tr>
<td>0x08D0</td>
<td>0000 9BDF</td>
</tr>
<tr>
<td>0x08D4</td>
<td>0000 0246</td>
</tr>
<tr>
<td>0x08D8</td>
<td>0000 8ACE</td>
</tr>
</tbody>
</table>
Pointers

Where else are they used?

- Provide method to pass arguments by reference to functions
- Provide method to pass more than one piece of information out of a function
- Another means of accessing arrays and dealing with strings
- Used in conjunction with dynamic memory allocation (creating variables at runtime)
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
```
CMPE-013/L

Introduction to "C" Programming

Maxwell James Dunne
Spring 2015
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 *:

\[ p = \& x; \]

• 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;

- \texttt{p->re} represents the data pointed to by \texttt{p}
  - \texttt{p->re} may be used anywhere you would use \texttt{x.re}
  - \texttt{->} is the structure dereference operator, equivalent to \texttt{(*p).re}
  - In the pointer declaration, the only significance of \texttt{*} 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;
}
```
Pointers
How Pointers Work

Example

```c
{  
    int x, y;
    int *p;
    
    x = 0xDEAD;
    y = 0xBEEF;
    p = &x;
    
    *p = 0x0100;
    p = &y;
    *p = 0x0200;
}
```
```c
{  
    int x, y;
    int *p;
    x = 0xDEAD;
    \textcolor{yellow}{y = 0xBEEF;}
    p = &x;
    \textcolor{yellow}{*p = 0x0100;}
    p = &y;
    \textcolor{yellow}{*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>
</tr>
</thead>
<tbody>
<tr>
<td>0x08B8</td>
</tr>
<tr>
<td>0x08BC</td>
</tr>
<tr>
<td>0x08C0</td>
</tr>
<tr>
<td>0x08C4</td>
</tr>
<tr>
<td>0x08C8</td>
</tr>
<tr>
<td>0x08CC</td>
</tr>
<tr>
<td>0x08D0</td>
</tr>
<tr>
<td>0x08D4</td>
</tr>
</tbody>
</table>

Variable at Address

<table>
<thead>
<tr>
<th>X</th>
<th>0000 0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>0000 BEEF</td>
</tr>
<tr>
<td>P</td>
<td>0000 08BC</td>
</tr>
<tr>
<td></td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>0000 0000</td>
</tr>
<tr>
<td></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 0100</th>
<th>0000 BEEF</th>
<th>0000 08C0</th>
<th>0000 0000</th>
<th>0000 0000</th>
<th>0000 0000</th>
<th>0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</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>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Maxwell James Dunne – Spring 2015
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};

int *p;
*p = x[0];
*p = x[0];  // Edit: Corrected to point to x[0]
*p++;
*(p + 1) = x[1];
```

- 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 (0\textsuperscript{th}) 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>0000 0001</th>
<th>0000 0002</th>
<th>0000 0003</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td>FFFF FFFF</td>
<td>FFFF FFFF</td>
<td></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>
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;
```
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;
```
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

```
float a[0] 0x0050
float a[1] 0x0054
float a[2] 0x0058
float a[3] 0x005C
float a[4] 0x0060
float a[5] 0x0064
float a[6] 0x0068
float a[7] 0x006C
float a[8] 0x0070
```
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.
Pointer Arithmetic
Larger Jumps

Example

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

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

ptr += 6;
```

16-bit Data Memory Words
Pointers

Pointer Arithmetic

Example

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

    *p += 4;
    ++p;
    *p = 0xDEAD1234BEEF;
    ++p;
    *p = 0xF1D04321F00D;
    p -= 2;
    *p = 0xBAAD0000F00D1;
}
```

32-bit Data Memory (RAM)

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

```
0000 0000
0000 0001
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;
```
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 (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</td>
<td>0005</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>x[1]</td>
<td>1234</td>
<td>BEEF</td>
<td>0000</td>
<td>DEAD</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>x[2]</td>
<td>0000</td>
<td>0003</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

p = 0x0810
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)

- `x[0]`:
  - Address: 0x07FC
  - Value: 0000 0005

- `x[1]`:
  - Address: 0x0800
  - Value: 1234 BEEF
  - Address: 0x0804
  - Value: 0000 0000
  - Address: 0x0808
  - Value: 0000 0000
  - Address: 0x080C
  - Value: 0000 DEAD

- `x[2]`:
  - Address: 0x0810
  - Value: 4321 F00D
  - Address: 0x0814
  - Value: 0000 F1D0
  - Address: 0x0818
  - Value: 0000 0810

- `p`:
  - Address: 0x0810
  - Value: 0000 0810
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 = 0xBA0D0000F00D;
```
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><code>z = *(p++) ;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>is equivalent to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>z = *p ;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>p = p + 1 ;</code></td>
</tr>
<tr>
<td>* (p++)</td>
<td></td>
<td><code>z = (*p) ++ ;</code></td>
</tr>
<tr>
<td></td>
<td>Post-Increment data pointed to by Pointer</td>
<td>is equivalent to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>z = *p ;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>*p = *p + 1 ;</code></td>
</tr>
</tbody>
</table>
Pointers
Post-Increment / Decrement Syntax

Example

```
{  
    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>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0000 0001</td>
<td>0000 0002</td>
</tr>
<tr>
<td>0000 0003</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)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0001</td>
<td>0000 0002</td>
<td>0000 0003</td>
</tr>
<tr>
<td>0000 0800</td>
<td>0000 0006</td>
<td></td>
</tr>
<tr>
<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)++;
```

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 0001</td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td>0000 0003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0000 0006</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

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 0000 0000
0x0800 0000 0001
0x0804 0000 0002
0x0808 0000 0003
0x0810 0000 0804
0x0814 0000 0007
0x0818 0000 0000

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 Pointer</td>
<td>z = * (++p);</td>
</tr>
<tr>
<td></td>
<td>is equivalent to:</td>
<td>p = p + 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>z = *p;</td>
</tr>
<tr>
<td>*(++p)</td>
<td>Pointer</td>
<td>z = ++(*p);</td>
</tr>
<tr>
<td></td>
<td>is equivalent to:</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

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

32-bit Data Memory (RAM)

Address 0x7FCE

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FCE</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x8000</td>
<td>0000 0001</td>
</tr>
<tr>
<td>0x8004</td>
<td>0000 0002</td>
</tr>
<tr>
<td>0x8008</td>
<td>0000 0003</td>
</tr>
<tr>
<td>0x800C</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x8010</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x8014</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x8018</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);
}
```

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>
</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 0002</td>
<td>0000 0003</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804</td>
<td>0000 0007</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0808</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

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 0003</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td>0x0800</td>
<td>0x0804</td>
<td></td>
</tr>
</tbody>
</table>
```

Remember:

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

Pre-Increment / Decrement Syntax

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

**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 0008</td>
</tr>
<tr>
<td>0x0810</td>
<td>0000 0000</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- 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
• This will prevent it from unintentionally corrupting a memory location if it is accidentally used before it is initialized

Example

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

**Note:**

- `NUL` is the character `'\0'`
- `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 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.

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08C0</td>
<td>0000</td>
</tr>
<tr>
<td>0x91C0</td>
<td>91C0</td>
</tr>
<tr>
<td>0x91C4</td>
<td>4950</td>
</tr>
<tr>
<td></td>
<td>3343</td>
</tr>
<tr>
<td></td>
<td>0058</td>
</tr>
<tr>
<td></td>
<td>4D32</td>
</tr>
</tbody>
</table>

Maxwell James Dunne – Spring 2015
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.
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";
```
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)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9D00 3FC0</td>
<td>9D00 3FC3</td>
<td>9D00 3FC7</td>
<td>9D00 3FCC</td>
</tr>
</tbody>
</table>

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 an 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>
<th>3FA4-3FA8</th>
<th>3FA9-3FAD</th>
<th>3FAE-3FB2</th>
<th>3FB3-3FB7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>On</td>
<td>Off</td>
<td>Main</td>
<td>Aux</td>
</tr>
<tr>
<td>0000 6E4F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6666 4F00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>614D 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4100 6E69</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 7875</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

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);

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));

Heap (top)

NULL
**Dynamic Memory**

**NULL pointers**

**Example**

```c
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);
}
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);
```
// 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!