CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
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 \n(count-1) chars are received
- **stream** is stdin
Text I/O
fgets() example

```
#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 `strncpy()`!)*
    - 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**

```c
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

\texttt{strtok()}\n
**Syntax**

\begin{verbatim}
char *strtok(char *s1, const char *s2);
\end{verbatim}

- \texttt{s1} (input/output), string to be tokenized
  - Will be modified!
- \texttt{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!";
```

```
This is an example!
```

Maxwell James Dunne

CMPE-013/L: “C” Programming
String Processing

strtok() Details

Example

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

firstToken
s1

This is an example !
String Processing

strtok() Details

Example

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

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

firstToken

s1

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, " ");
```

```
firstToken
s1
```

```
th is 
```

```
thirdToken
```

```
t is an 
```

```
secondToken
```

```
 an example!
```

```
example!
```
String Processing

strtok() Details

Example

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

```
This is an example!
```

```
This
is
an
example!
```

- firstToken
- secondToken
- thirdToken
- fourthToken

Maxwell James Dunne
4 <= 5.56 + 1
input

atof(input) -> stack

pop
item = items[curindex]

curindex --
Stack push

rest = stack push(1)

if (resp == ERROR)
    print error

if (stack push())
    struct (NULL)

\[ \text{Stack} \begin{array}{c}
\text{rest} = \text{stack push}(1) \\
\text{if}(\text{resp} == \text{ERROR}) \\
\text{print error} \\
\text{if}(\text{stack push}()) \\
\text{struct} (\text{NULL})
\end{array} \]
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 = "asdfsasdf\0"
```
Hardware

0

Shut down 6 times
String Processing

`strncat()`

**Syntax**

```
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...
Pointers
What are pointers?

- A pointer holds the address of another variable or function

32-bit Data Memory (RAM)

Variable at Address

Integer Variable: \( x \)

0000 0123

0000 0804

Pointer Variable: \( p \)

0x0800

0x0808

0x0810

0x0814
Pointers
What do they do?

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

\[ p = 0x0804 ; \]

**Direct Access via x**
\[ x = 0x0123 ; \rightarrow x \]

**Indirect Access via *p**
\[ *p = 0x0123 ; \rightarrow p \]

`p` 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)
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)
Points
How to Create a Pointer Variable

Syntax

```
type *ptrName;
```

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

Example

```
int *iPtr;      // Create a pointer to int
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
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

```
type *ptrName;
```

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

Example

```
int *iPtr;       // Create a pointer to int
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 \underline{without} the dereference operator *:

\[
p = \&x;
\]

• This assigns the address of the variable \underline{x} to the pointer \underline{p} (\underline{p} now points to \underline{x})

\underline{Note:} \underline{p} must be declared to point to the type of \underline{x} (e.g. int \underline{x}; int *\underline{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

```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)

CMPE-013/L: “C” Programming
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;

- \textbf{p->re} represents the data pointed to by \textbf{p}
  - \textbf{p->re} may be used anywhere you would use \textbf{x.re}
  - \textbf{->} is the structure dereference operator, equivalent to \textbf{(*p).re}
  - In the pointer declaration, the only significance of \textbf{*} 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;
}
```
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</th>
<th>Address</th>
<th>32-bit Data Memory (RAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0x08B8</td>
<td>0000 0000</td>
</tr>
<tr>
<td>y</td>
<td>0x08BC</td>
<td>0000 0000</td>
</tr>
<tr>
<td>p</td>
<td>0x08C0</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>0x08C4</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>0x08C8</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>0x08CC</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>0x08D0</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>0x08D4</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>
```
Pointers
How Pointers Work

Example

```c
{ int x, y;
  int *p;
  x = 0xDEADB;    // Set x to 0xDEADB
  y = 0xBEEF;
  p = &x;
  *p = 0x0100;    // Set the address of x to 0x0100
  p = &y;
  *p = 0x0200;    // Set the address of y to 0x0200
}
```

32-bit Data Memory (RAM)

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

x

y

p

Address

0x08B8
0x08BC
0x08C0
0x08C4
0x08C8
0x08CC
0x08D0
0x08D4
Pointers
How Pointers Work

Example

```c
{  
    int x, y;
    int *p;
    x = 0xDEAD;
    y = 0xBEFF;
    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>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>y</td>
<td>DEAD</td>
<td>BEEF</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>p</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>
Pointers
How Pointers Work

Example

```c
{  
    int x, y;
    int *p;

    x = 0xDEAD;
    y = 0xBEFF;
    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>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0000</td>
<td>DEAD</td>
</tr>
<tr>
<td>y</td>
<td>0000</td>
<td>BEEF</td>
</tr>
<tr>
<td>p</td>
<td>0000</td>
<td>08BC</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;
}
```

32-bit Data Memory (RAM)

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

Variable at Address

- `x`: 0x0100
- `y`: 0x0200
- `p`: 0x08BC
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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x08B8</td>
<td>0x08BC</td>
<td>0x08C0</td>
<td>0x08C4</td>
</tr>
<tr>
<td></td>
<td>0x08C8</td>
<td>0x08CC</td>
<td>0x08D0</td>
<td>0x08D4</td>
</tr>
</tbody>
</table>

Variable at Address

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 and Arrays
A Quick Reminder...

- Array elements occupy consecutive memory locations

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

int *y = &x[0];  
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
</tr>
<tr>
<td>0x0800</td>
</tr>
<tr>
<td>0x0804</td>
</tr>
<tr>
<td>0x0808</td>
</tr>
<tr>
<td>0x080C</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFF</td>
<td>FFFF</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td>FFFF</td>
<td>FFFF</td>
<td></td>
</tr>
</tbody>
</table>

- 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
```

```c
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>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td>FFFF</td>
</tr>
<tr>
<td>0x0800</td>
<td>0000</td>
</tr>
<tr>
<td>0x0804</td>
<td>0001</td>
</tr>
<tr>
<td>0x0808</td>
<td>0002</td>
</tr>
<tr>
<td>0x0810</td>
<td>0003</td>
</tr>
</tbody>
</table>

- More on this in just a bit...
Pointers and Arrays
A Preview of Pointer Arithmetic

- Incrementing a pointer will move it to the next element of the array

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

- More on this in just a bit...
Pointers and Arrays
A Preview of Pointer Arithmetic

- Incrementing a pointer will move it to the next element of the array

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

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</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 0804</td>
</tr>
</tbody>
</table>

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

Incrementing Pointers

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

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

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

Incrementing Pointers

Example

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

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

32-bit Data Memory Words
Pointer Arithmetic
Larger Jumps

- Adding or subtracting any other number with the pointer will change it by a multiple of the number of bytes of its type.
- If we have

```c
short int x;
short int *p = &x;
p += 3;
```

We will get the address of `p` incremented by 6 since a `short int` variable occupies 2 bytes of memory.
Pointer Arithmetic

Larger Jumps

Example

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

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

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

Pointer Arithmetic

Example

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

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

Pointer Arithmetic

Example

```c
{ 
    long long x[] = {1, 2, 3};
    long long *p = x;
    *p += 4;
    ++p;
    *p = 0xDEAD1234BEEF;
    ++p;
    *p = 0xF1D04321F00D;
    p -= 2;
    *p = 0.BAD0000F00D1;
}
```
Pointers
Pointer Arithmetic

Example

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

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

32-bit Data Memory (RAM)

```
0000 0000
0000 0005
0000 0000
0000 0000
0000 0002
0000 0000
0000 0003
0000 0808
```
Pointers

Pointer Arithmetic

Example

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

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 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 0005</td>
</tr>
<tr>
<td>0x0800</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0804</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0808</td>
<td>1234 BEEF</td>
</tr>
<tr>
<td>0x080C</td>
<td>0000 DEAD</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 0808</td>
</tr>
</tbody>
</table>

Maxwell James Dunne
CMPE-013/L: "C" Programming
Pointers

Pointer Arithmetic

Example

```c
#include <stdio.h>

int main() {
    long long x[] = {1, 2, 3};
    long long *p = x;

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

    printf("%d, 0x%08x, 0x%08x, 0x%08x\n", x[0], x[1], x[2], p);
    return 0;
}
```

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 0000 005</td>
<td>1234 BEEF</td>
<td>0000 0003</td>
</tr>
<tr>
<td>0x0800</td>
<td>0000 0000 000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0804</td>
<td>0000 0000 000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0808</td>
<td>0000 0000 000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x080C</td>
<td>0000 0000 000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0810</td>
<td>0000 0000 000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0814</td>
<td>0000 0000 000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0818</td>
<td>0000 0000 000</td>
<td>0000 0000</td>
<td>0000 0000</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 = 0.BAD0000F00D1;
}
```

32-bit Data Memory (RAM)

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

- `x[0]`: 0000 0005
- `x[1]`: 1234 BEEF
- `x[2]`: 4321 F00D

- `p`: 0000 0810
# Pointers

**Pointer Arithmetic**

```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)

Address

0x07FE

0x0800

0x0802

0x0804

0x0806

0x0808

0x080A

0x080C

0x0000 0000

0x0000 BAD0

0x0000 0000

0x000F 00D1

0x0000 0000

0x0000 BAD0

0x1234 BEEF

0x0000 DEAD

0x0000 F1D0

0x4321 F00D

0x0000 0800
Pointsers

Post-Increment/Decrement Syntax Rule

- Care must be taken with respect to operator precedence when doing pointer arithmetic:

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

Post-Increment / Decrement Syntax

Example

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

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000 0000</td>
<td>0000 0001</td>
</tr>
<tr>
<td></td>
<td>0000 0002</td>
<td>0000 0003</td>
</tr>
<tr>
<td>p</td>
<td>0000 0800</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0000 0000</td>
<td></td>
</tr>
</tbody>
</table>

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

Post-Increment / Decrement Syntax

Example

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

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0000 0001</td>
</tr>
<tr>
<td>x[1]</td>
<td>0000 0002</td>
</tr>
<tr>
<td>x[2]</td>
<td>0000 0003</td>
</tr>
<tr>
<td>p</td>
<td>0000 0800</td>
</tr>
<tr>
<td>y</td>
<td>0000 0006</td>
</tr>
</tbody>
</table>

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

Post-Increment / Decrement Syntax

Example

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

32-bit Data Memory (RAM)

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

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

Remember: `*(p++)` is the same as `*p++`
Pointers

Post-Increment / Decrement Syntax

Example

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

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
<th>0x0804</th>
<th>0x0808</th>
<th>0x080C</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[0]</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>x[1]</td>
<td>0000</td>
<td>0000</td>
<td>0003</td>
<td>0003</td>
<td>0000</td>
</tr>
<tr>
<td>x[2]</td>
<td>0000</td>
<td>0000</td>
<td>0003</td>
<td>0003</td>
<td>0000</td>
</tr>
<tr>
<td>p</td>
<td>0000</td>
<td>0804</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>y</td>
<td>0000</td>
<td>0007</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

Remember:

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

Pre-Increment/Decrement Syntax 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></td>
<td></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</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer</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**

```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>
<th>0000 0804</th>
<th>0000 0808</th>
<th>0000 080C</th>
<th>0000 0810</th>
<th>0000 0814</th>
<th>0000 0818</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0808</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x080C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Remember:

\[
* (++p) \text{ is the same as } *++p
\]
Pointers
Pre-Increment / Decrement Syntax

Example

```
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>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0000 0000</td>
<td>0000 0001</td>
<td>0000 0002</td>
<td>0000 0003</td>
<td>0000 0804</td>
</tr>
<tr>
<td>x[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0000 0007</td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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);
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
<th>0x0804</th>
<th>0x0808</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td></td>
<td>0000</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td></td>
<td>0000</td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td></td>
<td>0000</td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td>0000</td>
<td>0804</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td>0000</td>
<td>0007</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);  
}
```

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;
```

**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 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);
}
```
complex \( X \), \( \text{re} \)

complex \( \ast Q \)

\( Q = \mathbb{Q} \times \)

\( Q \rightarrow \text{re} = 4 \).
Pointers and Strings

- So far, we have worked with strings strictly as arrays of \texttt{char}.
- Strings may be created and used with pointers much more elegantly.

String declaration with a pointer:

\begin{verbatim}
char *str = "PIC32MX"; str
\end{verbatim}

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";
```

<table>
<thead>
<tr>
<th>RAM</th>
<th>ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9D0008C0</td>
<td>0x9D0008C0</td>
</tr>
<tr>
<td>0xA0000FB0</td>
<td>0xA0000FC0</td>
</tr>
<tr>
<td>0xA0000FB4</td>
<td>0xA0000FC1</td>
</tr>
<tr>
<td>0xA0000FB5</td>
<td>0xA0000FC2</td>
</tr>
<tr>
<td>0xA0000FB6</td>
<td>0xA0000FC3</td>
</tr>
<tr>
<td>0xA0000FB7</td>
<td></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.

```c
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>
<tr>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</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>p[0]</td>
<td>On \0 \0 \0</td>
<td>Off \0 \0</td>
<td>Main \0</td>
<td>Aux \0</td>
</tr>
<tr>
<td>0000 0000</td>
<td>0000 6E4F</td>
<td>6666 4F00</td>
<td>614D 0000</td>
<td>4100 6E69</td>
</tr>
<tr>
<td>p[1]</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>p[2]</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>p[3]</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></td>
<td>0000 0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 0000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Maxwell James Dunne
Arrays of Pointers

Dereferencing

- To use the value pointed to by a pointer array element, just dereference it like you would an ordinary variable:

  \[ y = \*p[0]; \]

- Using \( \*p[0] \) is the same as using the object it points to, such as \( x \) or the string literal "My String" from the previous slide.
Arrays of Pointers
Accessing Strings

Example

```c
int i = 0;
char *str[] = {"Zero", "One", "Two", "Three", "Four", "\0"};

int main(void)
{
    while(*str[i] != '\0') {
        printf("%s\n", str[i++]);
    }
    while(1);
}
```
Dynamic Memory

malloc()

free()
Dynamic Memory

Rationale

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

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

```
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\n\t\t\t\t\t\t\t\t\t\tre:%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

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

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

NULL pointers

Example

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

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

NULL pointers

Example

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

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

`free()`

**Syntax**

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

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

**Example**

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

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

Invalid Free(ing)

Example

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

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

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

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