

ELEC / COMP 177 – Fall 2012

# Computer Networking

## → C Programming Essentials

Some slides from Kurose and Ross, *Computer Networking*, 5<sup>th</sup> Edition

# Experience

- Prior experience in programming languages
  - C++ programming?
  - Java programming?
  - C programming?
  - Other languages?
- Prior experience in Linux / FreeBSD / Unix
  - Yes?
  - No?

# C Programming

For C++ Programmers

# C++ Features Not in C

- No **classes** / object-oriented programming
- No **new** / **delete**
- No stream operators (<< and >>), cin, cout, ...
- No C++ Standard Libraries (e.g. iostream)
- `bool` keyword
  - Added back in C99 standard  
`#include <stdbool.h>`
- Declare variables anywhere inside function
  - Added back in C99 standard

# Output with printf()

- `printf("This is a string\n");`
- `printf("The integer is %i\n", num);`
- `printf("The floating-point values are %g and %g\n", num1, num2);`
- Lots of formatting options
  - # of digits before / after decimal point?
  - Pad with zeros?

# Output with printf()

Format "Type" Code	Corresponding Variable Type
d or i	int (interpret as signed 2's comp)
u	int (interpret as unsigned)
x	int (print as hexadecimal)
f or g	float/double
c	char
s	string (null-terminated array of chars)

*Prefix with l or ll (i.e. "long" or "long long" for larger 64-bit data types)*

# Input with scanf()

- Input from console
- `scanf ("%d %c", &myint, &mychar)`
- Requires the **address** of the destination variable
  - Use the `&` operator to obtain address
- `char myArray[256];`  
`scanf ("%255s", myArray);`
  - Note here that "myArray" is already a pointer...

# Documentation

- Man(ual) pages exist for common programming functions too
- `unix> man printf`
- `unix> man scanf`
- Or just google...



# Structures

```
struct database
{
    int id_number;
    int age;
    float salary;
};
```

```
int main()
{
    database employee;
    employee.age = 22;
    employee.id_number = 1;
    employee.salary = 12000.21;
}
```

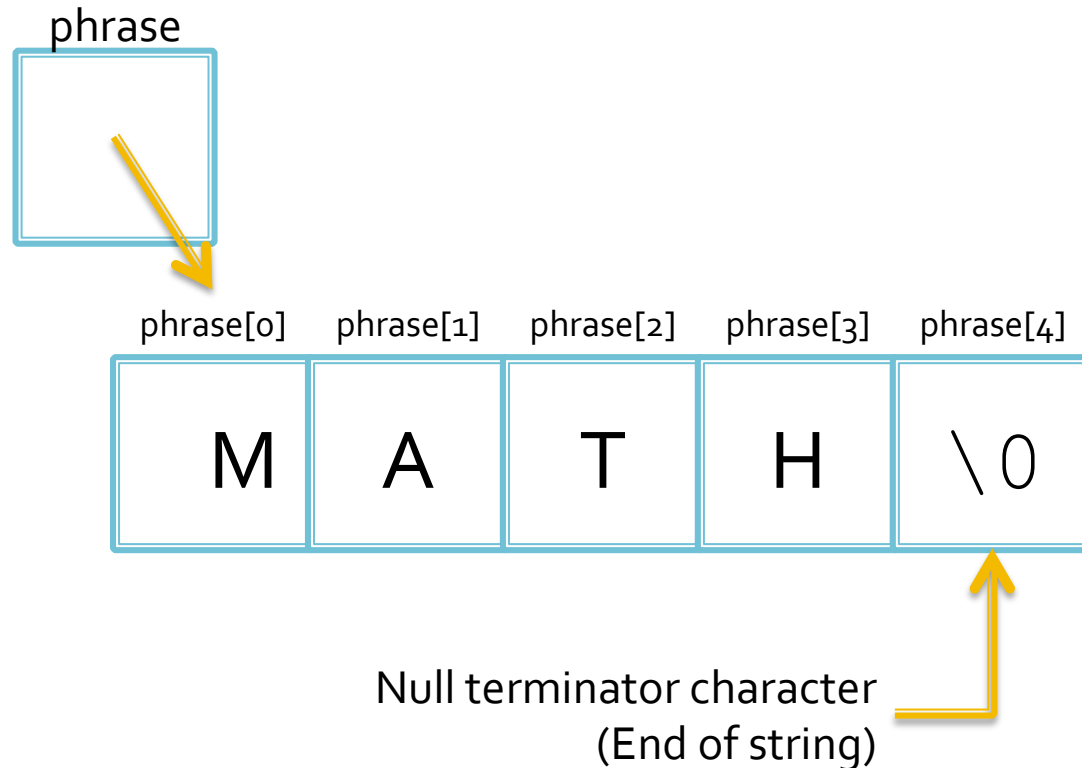
# C-Strings (Arrays of Characters)

# C Strings

- There is no such thing as a “string” in C!
- What do you get? **An array of characters**
  - Terminated by the null character ' \0 '
- Must manipulate element by element...
  - Not enough room in the array? Need a bigger array

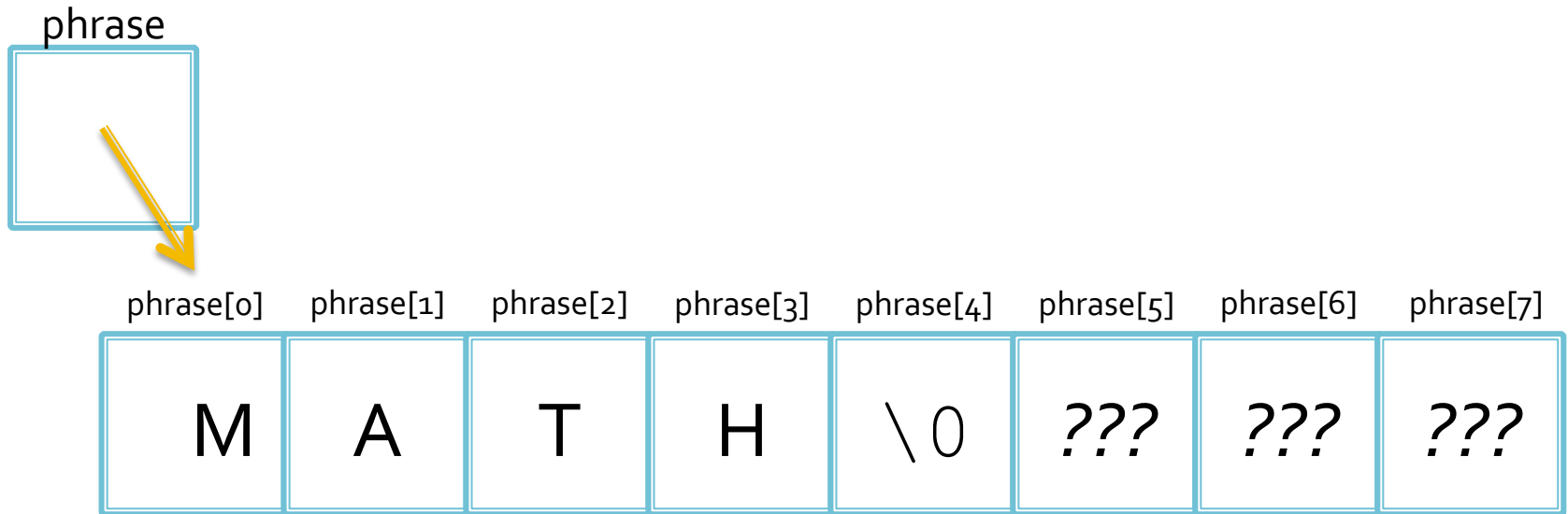
# Arrays of Characters

- `char phrase[] = "Math";`



# Arrays of Characters

■ `char phrase[8] = "Math";`



```
printf("%s\n", phrase);
```

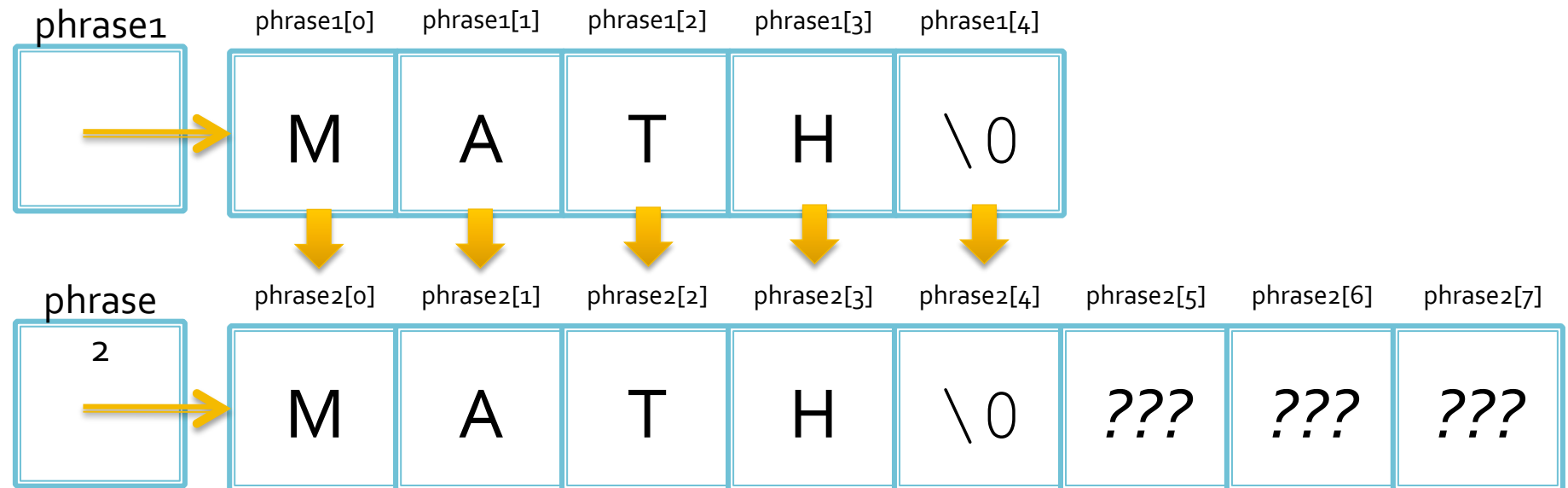
**Prints until it reaches  
the `\0` character!**

# Helpful Library for Character Arrays

- **#include <string.h>**
- Useful functions
  - strcpy - String copy
  - strcmp - String compare
  - strlen - String length
  - strcat - String concatenate

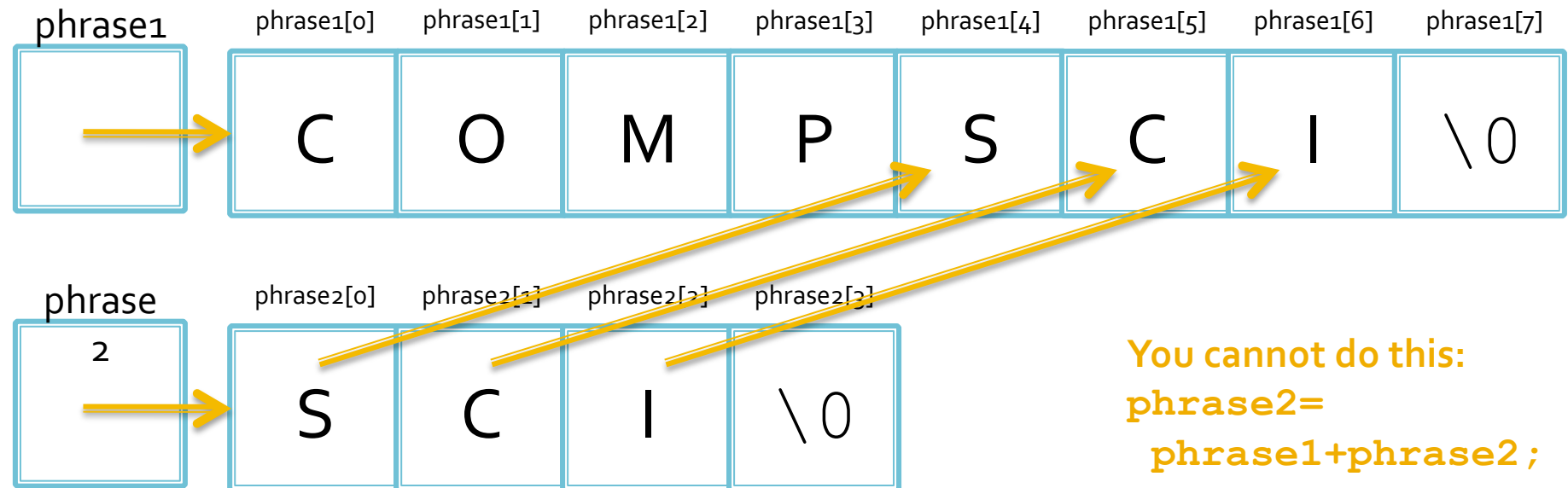
# String Copy

- `char phrase1[] = "Math";`
- `char phrase2[8];`
- `strcpy(phrase2, phrase1);`



# String Concatenation

- `char phrase1[8] = "Comp";`
- `char phrase2[] = "Sci";`
- **`strcat(phrase1, phrase2);`**



You cannot do this:  
`phrase2 =  
phrase1 + phrase2;`



# cctype Library

- Useful for character manipulation
- `#include <cctype.h>`
- **`toupper(char)` / `tolower(char)`** –  
Converts character to uppercase or lowercase
  - Example:

```
char c = toupper('a');  
printf("%c", c);    // A
```

# cctype Library

- **isalpha(char)** – Is the character a letter?
- **isdigit(char)** – Is the character a number 0-9?
- **isspace(char)** – Is the character whitespace? (space or newline character)
- **ispunct(char)** – Is the character punctuation? (technically, a visible character that is not whitespace, a letter, or a number)
- ... and several other variations

# Memory Management

# Memory Allocation with malloc()

- `#include <stdlib.h>`
- `void * malloc(int size)`
  - **Allocate** region in memory (aka “new”)
  - Argument: Size of region in bytes to allocate
  - Return value: Pointer to the region
- `void free(void * ptr)`
  - **De-allocate** region in memory (aka “delete”)
  - Argument: Pointer to the region

# Memory Allocation with malloc()

- `void * calloc(int count, int size)`
  - Basically the same as malloc!
    - Imagine you want an array of elements...
  - Argument 1: # of elements to allocate
  - Argument 2: Size of each element in bytes
  - Return value: Pointer to the region

# Memory Allocation with malloc()

- `void * realloc(void *ptr, int size);`
  - **Resize** a dynamic region of memory
    - Note that it might **move** to a new address!
  - Argument: Pointer to the original region
  - Argument 2: Desired size in bytes of new region
  - Return value: Pointer to the new region
    - It might be at the same address if you made it smaller
    - It might be at a new address if you made it larger

# Memory Management

- **Who implemented `malloc()`?**
- **C Standard Library:** `#include <stdlib.h>`
- There are different C Standard Library implementations!
  - Android: Bionic
  - Apple: BSD-based / Proprietary
  - Microsoft: Proprietary C Runtime Library
  - Linux: GNU C Library (glibc)  
<http://www.gnu.org/software/libc/>

# Memory Management

- Where does the `malloc()` memory come from?
- The Heap:
  - A region of memory for dynamic memory allocation
  - Per-process – each program gets its own heap
  - Managed by `malloc()` and related functions
  - Different from the stack, which is for static variables (known at compile-time)



# Memory Management

- Why do we need to call `free()` after calling `malloc()`?
  - Memory leak
  - `malloc()` cannot re-use that space ever, because its internal bookkeeping still thinks that region is used
  - Will only be recovered upon terminating program
    - Operating system wipes out all the memory allocated to your process (stack, heap, etc...)

# Buffer Overflow Vulnerability

- **What is a buffer overflow bug?**
  - `char buf1[8];`  
`char buf2[8];`  
`strcat(buf1, "excessive");`
- End up overwriting two characters beyond buf1!

# Buffer Overflow Vulnerability

- **Why is a buffer overflow bug dangerous?**
- What is beyond my buffer in memory?
  - Other variables and data? (probably `buf2`)
  - The stack? (further out)
  - **The return address to jump to after my function finishes?**
- If app is running as administrator, attacker now has full access!

# Memory Management

- **Limitless opportunities in C** for errors regarding memory 😞
  - Forgetting to `free()` some dynamic memory
  - Trying to `free()` dynamic memory more than once
  - Losing a pointer to dynamic memory (memory is “lost”)
  - Accessing array elements past the end of the array
  - Mis-calculating array pointers that miss their desired target
- **Suggestion: Learn Valgrind to analyze your program and detect / trace errors**
  - See link on class *Resources* page

# What's the Error?

```
char *a = malloc(128*sizeof(char));  
char *b = malloc(128*sizeof(char));  
b = a;  
free(a);  
free(b);
```

<http://www.yolinux.com/TUTORIALS/C++MemoryCorruptionAndMemoryLeaks.html>

# What's the (Potential) Error?

```
char *a = malloc(128*sizeof(char));
```

```
dataLen = <some value...>
```

```
// Copy "dataLen" bytes  
// starting at *data to *a  
memcpy(a, data, dataLen);
```

<http://www.yolinux.com/TUTORIALS/C++MemoryCorruptionAndMemoryLeaks.html>

# What's the Error?

```
ptr = (char *) malloc(strlen(string_A));  
strcpy(ptr, string_A);
```

<http://www.yolinux.com/TUTORIALS/C++MemoryCorruptionAndMemoryLeaks.html>

# What's the Error?

```
int *get_ii()
{
    int ii = 2;    // Local stack variable
    return &ii;
}
main()
{
    int *ii;
    ii = get_ii();
    ... Do stuff using ii pointer
}
```

<http://www.yolinux.com/TUTORIALS/C++MemoryCorruptionAndMemoryLeaks.html>



OKAY, HUMAN.

HUH?

BEFORE YOU  
HIT 'COMPILE',  
LISTEN UP.



YOU KNOW WHEN YOU'RE  
FALLING ASLEEP, AND  
YOU IMAGINE YOURSELF  
WALKING OR  
SOMETHING,



AND SUDDENLY YOU  
MISSTEP, STUMBLE,  
AND JOLT AWAKE?

YEAH!



WELL, THAT'S WHAT A  
SEGFALT FEELS LIKE.

DOUBLE-CHECK YOUR  
DAMN POINTERS, OKAY?



# Memory Management

- **What's a NULL pointer?**
  - Pointer value is 0x00000000
  - *Meaning* is that the pointer is not pointing anywhere
- **What happens if you deference a NULL pointer?**
  - Telling the computer to read from (or write) to the value stored in the pointer, which is 0x00000000
  - Behavior undefined and generally unpleasant on various computer systems

# Memory Management

- **“Segfault” = Segmentation Fault**
- Your program tried to read or write a *virtual memory address* that is not allowed
  - Tried to read memory outside of program bounds?
  - Tried to write read-only memory regions? (used for program data)

# Memory Management

- “**Segmentation**” was the name of an old system (back before Intel 386 processors) used to divide physical computer memory into many virtual address regions, one per application process
  - The *Segfault* name stuck even though we now use **paging** to manage virtual memory

# Homework

# Homework 3

- **Discuss assignment**
  - What are we doing?
  - What are the requirements?
  - Discuss possible search algorithm (“pivot point”)
- **Where to get programming help?**
  - Class resources page:  
<http://ecs-network.serv.pacific.edu/ecpe-177/resources>
  - Friends
    - Honor code: Homework is done individually
      - Allowed: Planning solution strategies, helping each other to debug programs...
      - Not allowed: Copying code from classmates

# Lab Exercises

# Lab Exercises

---

- See Handout