



Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

C Programming

Lab Schedule

Activities

➤ This Week

- Intro to C
- Intro to Build Tools and Makefiles
- **Lab 3 – Build Tools**

➤ Next Week

- **Lab 4 – C Programming Project**

Deadlines

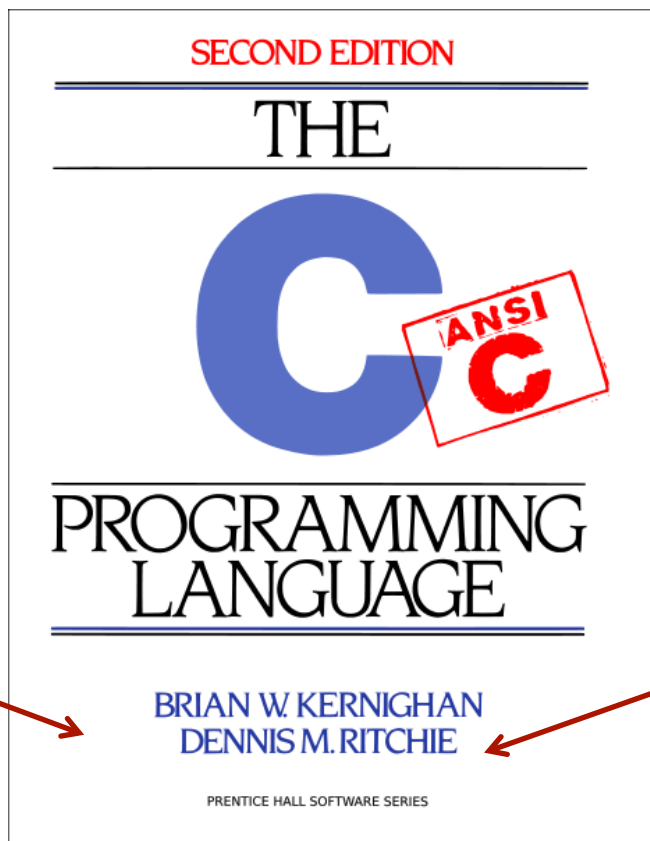
- **Lab 3 – Feb 2nd 2015
by 5am**
- **Lab 4 – Feb 16th 2015
by 5am**

Person of the Day: Dennis Ritchie



- Creator of **C programming language**
- Co-creator of **Unix**
(with Ken Thompson, Brian Kernighan, and others at Bell Labs)
- Winner of **ACM Turing Award**
- 9/9/1941—10/12/2011

Person of the Day: Dennis Ritchie



➤ *“Pretty much everything on the web uses those two things: C and UNIX. The browsers are written in C. The UNIX kernel — that pretty much the entire Internet runs on — is written in C. Web servers are written in C, and if they’re not, they’re written in Java or C++, which are C derivatives, or Python or Ruby, which are implemented in C. And all of the network hardware running these programs I can almost guarantee were written in C. It’s really hard to overstate how much of the modern information economy is built on the work Dennis did.”*

➤ Rob Pike, Bell Labs / Google



Dennis Ritchie and Ken Thompson use a teletypewriter to run a program on a UNIX-based computer system they co-founded at Bell Labs in New Jersey. Their development work more than 40 years ago facilitated the realization of the Internet.

C Programming



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 in C99 standard
- Declare variables anywhere inside function
 - Added 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);`

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)

- Lots of formatting options not listed here...
 - # of digits before / after decimal point?
 - Pad with zeros?

Input with scanf()

- Input from console
- `scanf("%d %c", &myint, &mychar)`
- Requires the **address** of the destination variable
 - Use the `&` operator to obtain address
- Caveat: Array names are already the “address of”!
 - ```
char myarray[8];
scanf("%s", myarray)
```



# Documentation

➤ **Man(ual) pages exist for common programming functions too**

➤ `unix> man printf`

➤ `unix> man scanf`

# Structures

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

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

# C-Strings (Arrays of Characters)

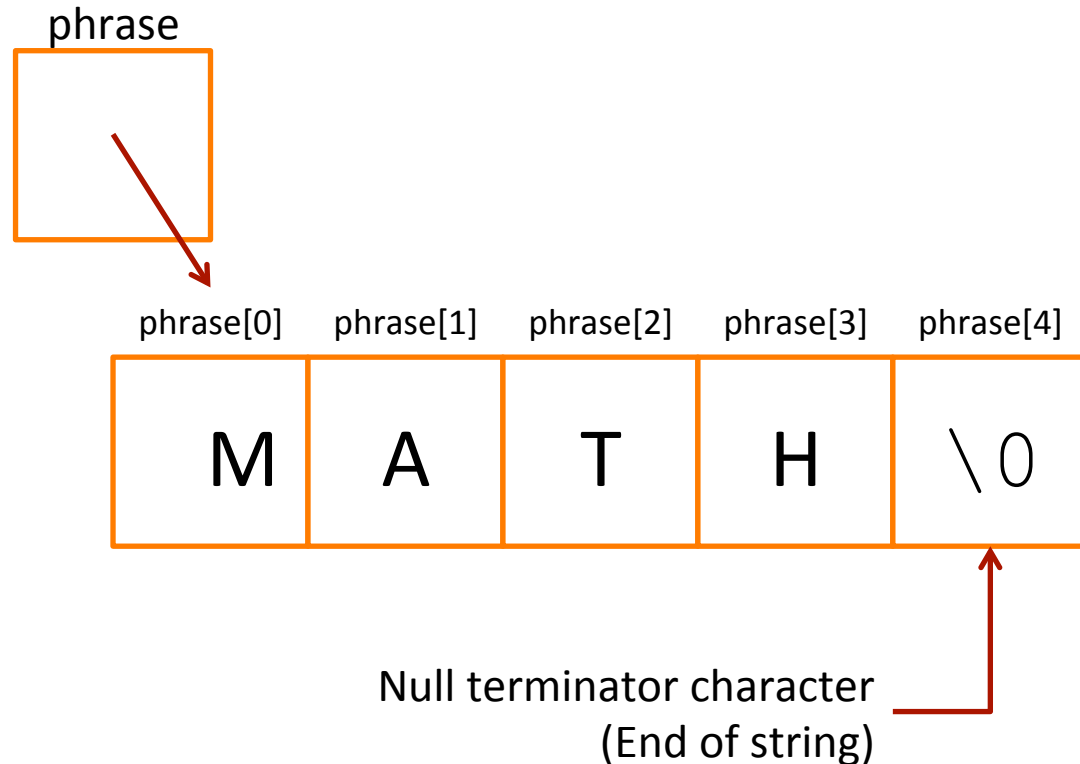


# C Strings

- **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

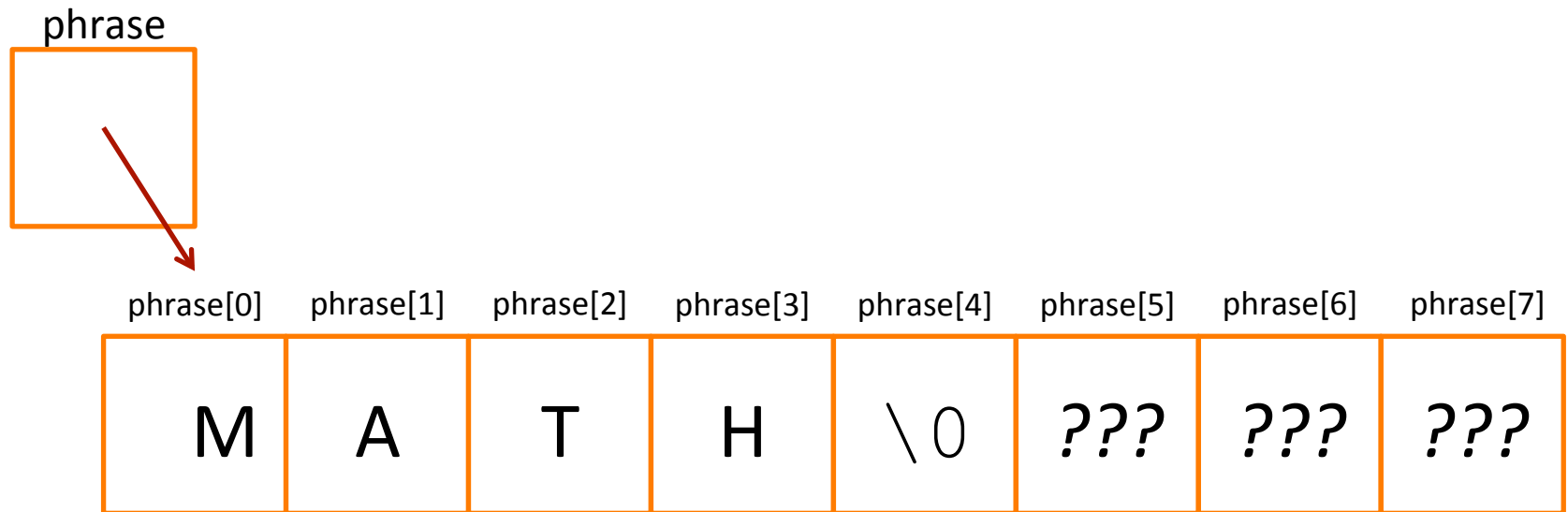
# Arrays of Characters

```
➔ 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

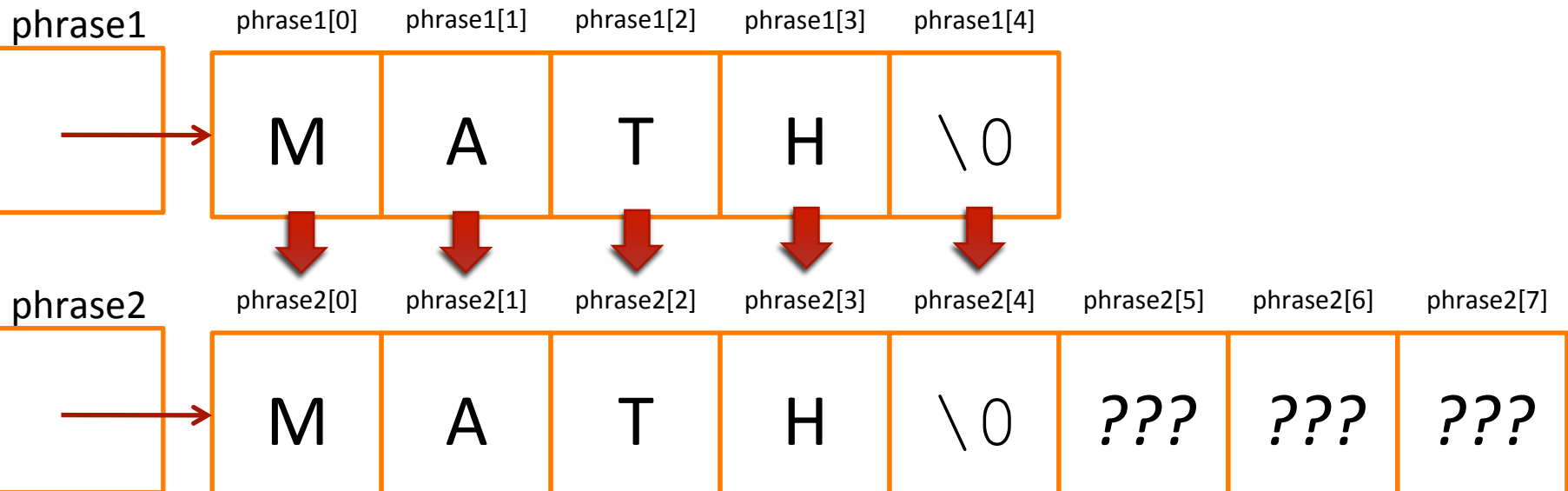
- `#include <string.h>`
- Useful functions
  - `strcpy` - String copy
  - `strcmp` - String compare
  - `strlen` - String length
  - `strcat` - String concatenate

# String Copy

➔ `char phrase1[] = "Math";`

➔ `char phrase2[8];`

➔ `strcpy(phrase2, phrase1);`

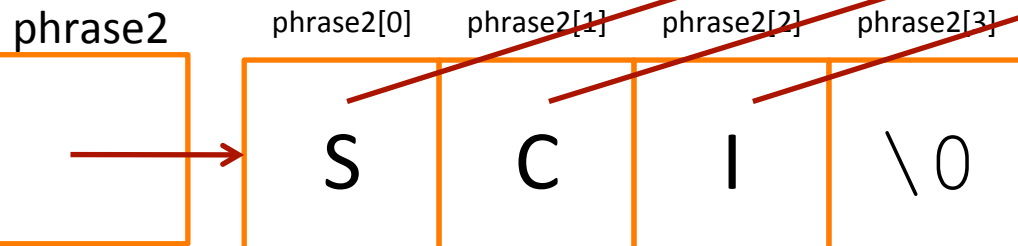
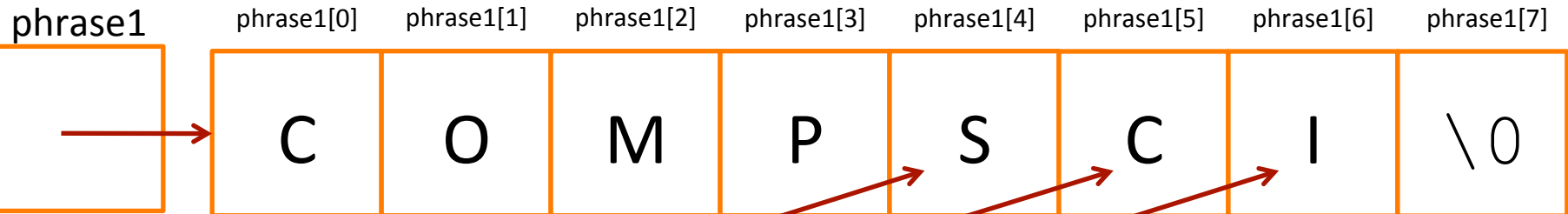


# String Concatenation

➔ `char phrase1[8] = "Comp";`

➔ `char phrase2[] = "Sci";`

➔ `strcat(phrase1, phrase2);`



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

# ctype Library

- Useful for character manipulation
- `#include <ctype.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

## ➤ `malloc()` outline:

1. Call `malloc()` and request memory
2. `malloc()` checks existing heap size
  - Sufficient? Update bookkeeping to mark space as “used” and return address to your program
  - Insufficient?
    1. **Call operating system** via `brk()` / `mmap()` to grow the heap (plus a little extra for future requests)
    2. Update bookkeeping and return address to your program

# 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...)

# Memory Management

- OS creates **virtual memory** space for process when started
- Region is huge (full 32 or 64 bit space)
  - **Not** fully mapped to physical memory
  - Otherwise you could only fit 1 program in memory

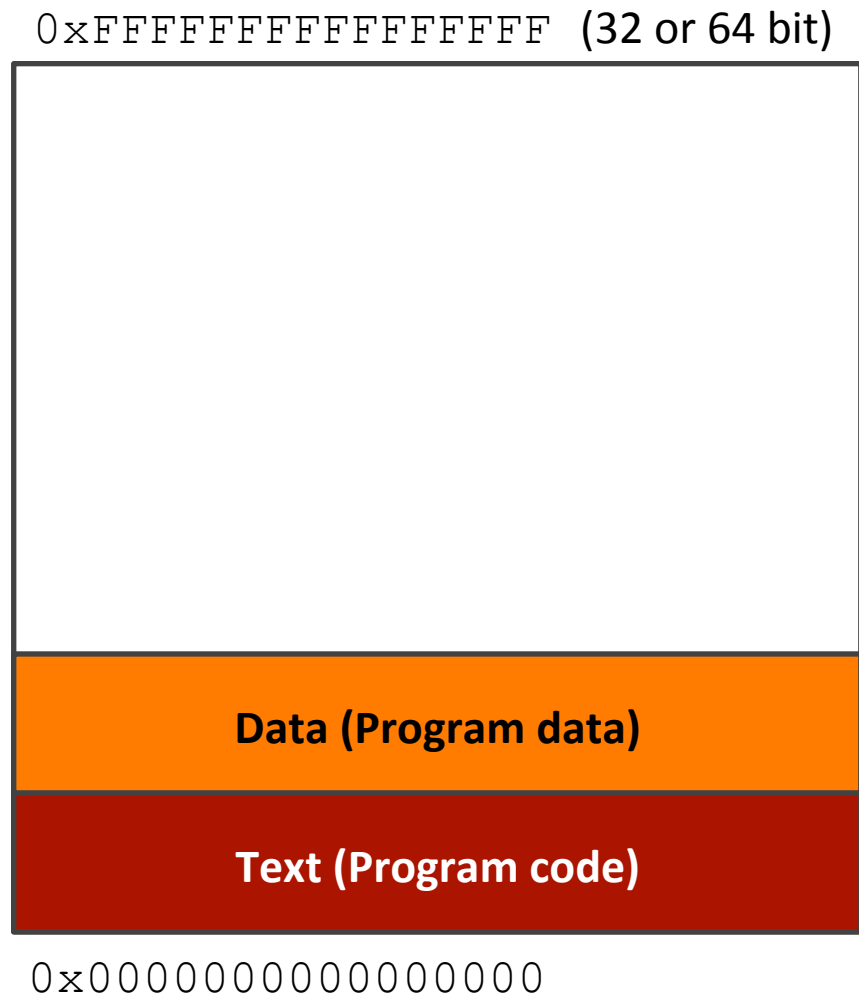
0xFFFFFFFFFFFFFFFF (32 or 64 bit)

*Virtual Memory Space  
for new process*

0x0000000000000000

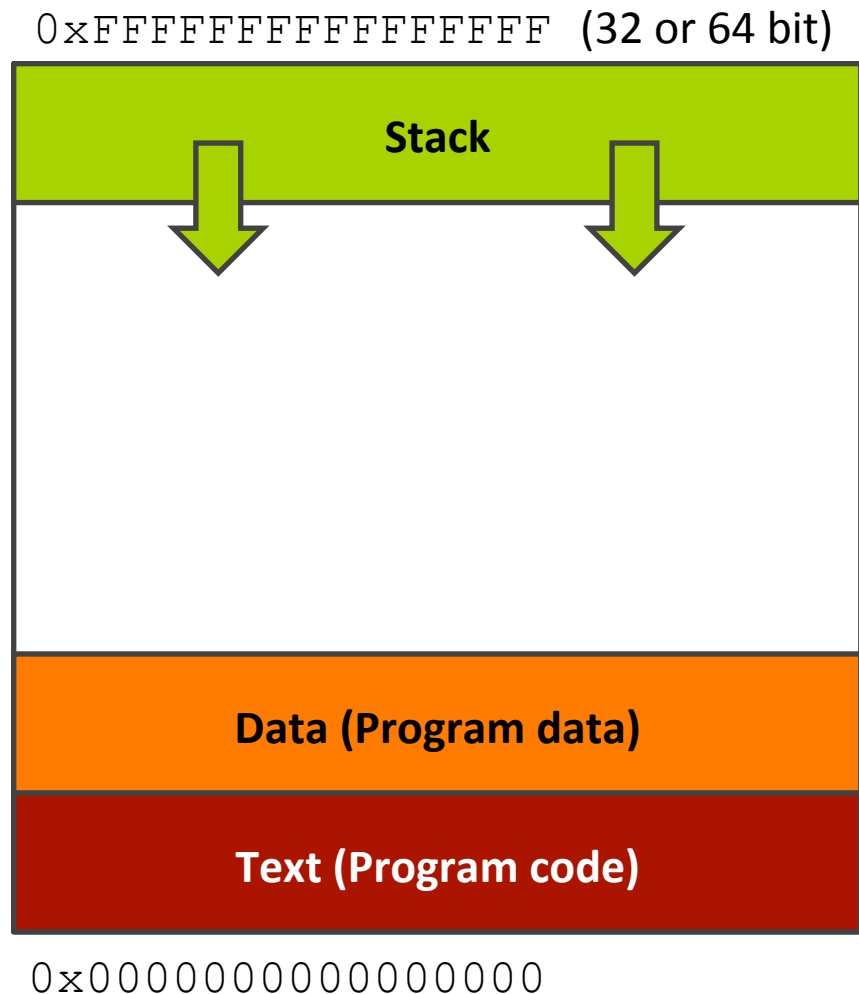
# Memory Management

- OS loads in the program from disk
- “Text” region
  - Program **code**
- “Data” region
  - Program fixed **data**



# Memory Management

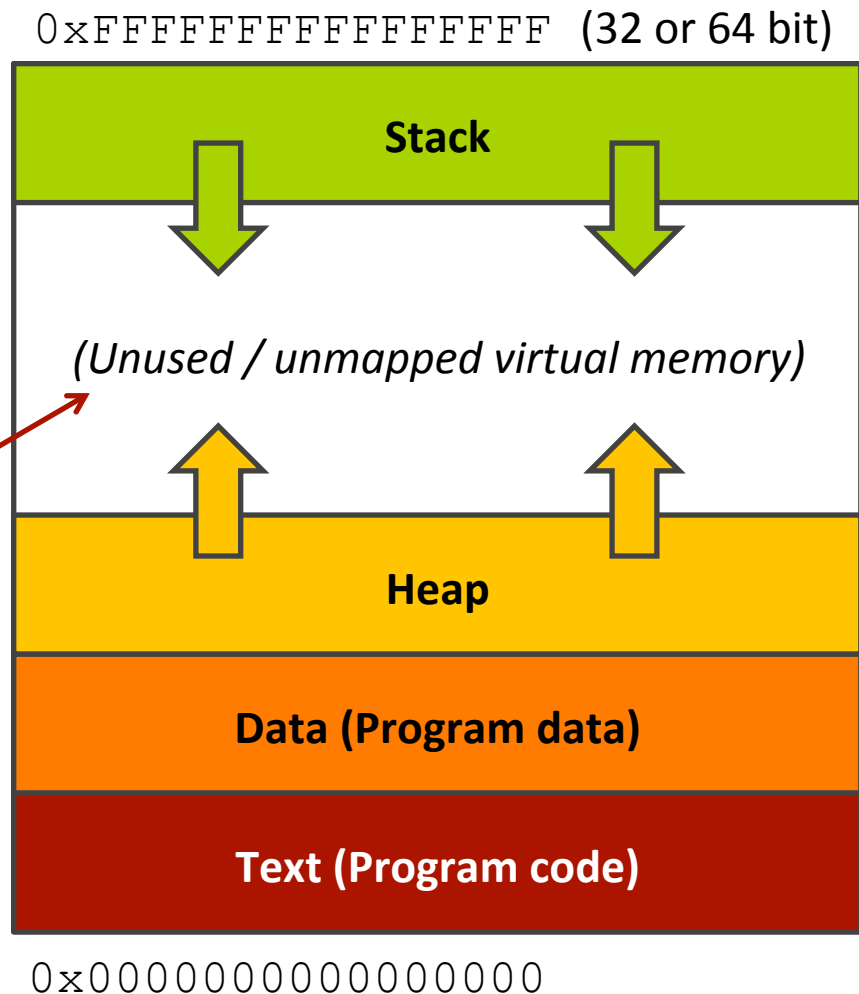
- **Stack** created to track program function calls and local variables





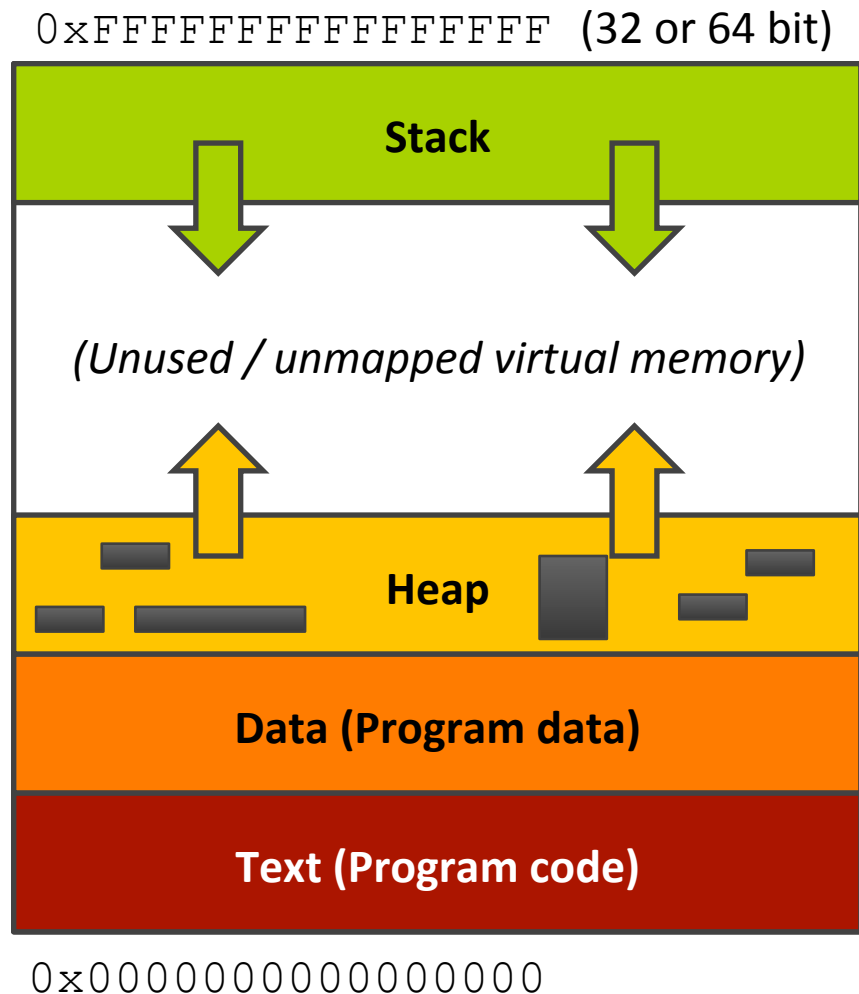
# Memory Management

- **Heap** created to store dynamic memory from `malloc()` and related functions
- Not to scale – this unused region is **huge!**



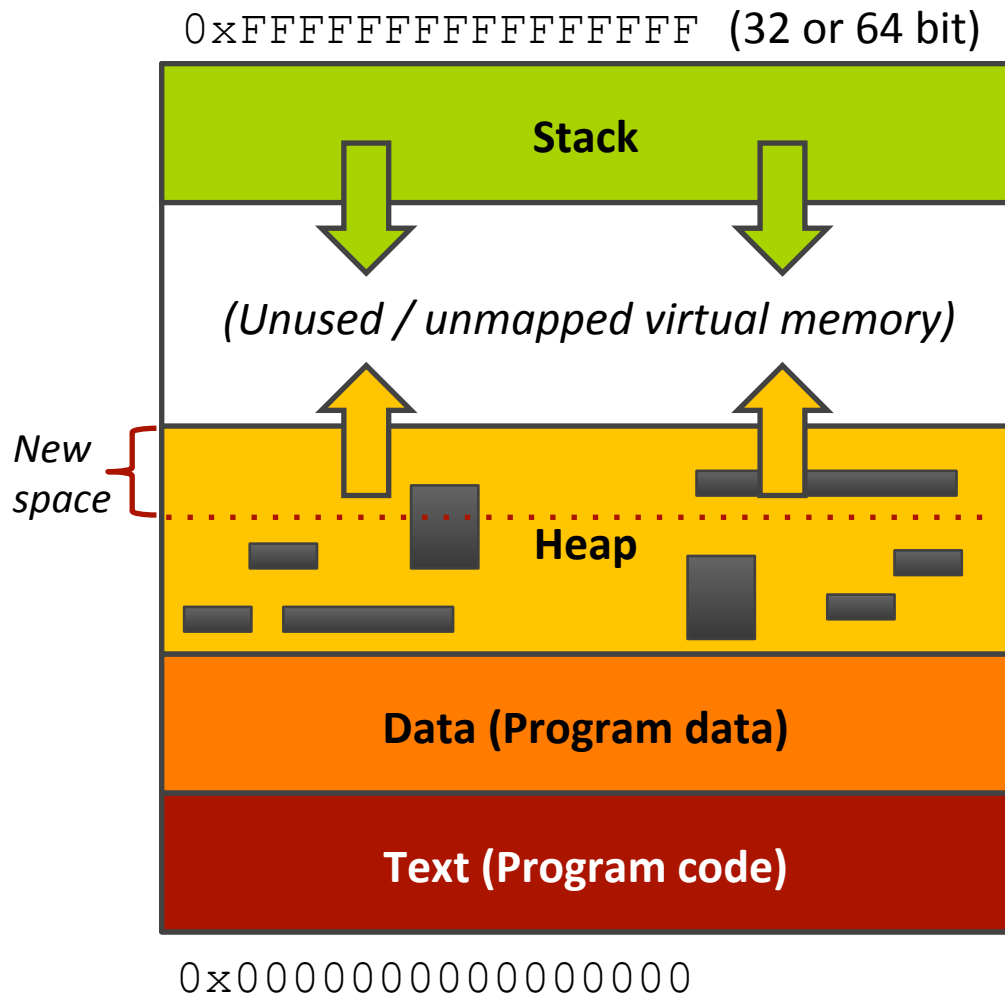
# Memory Management

- Program starts running
- `malloc()` allocates some memory



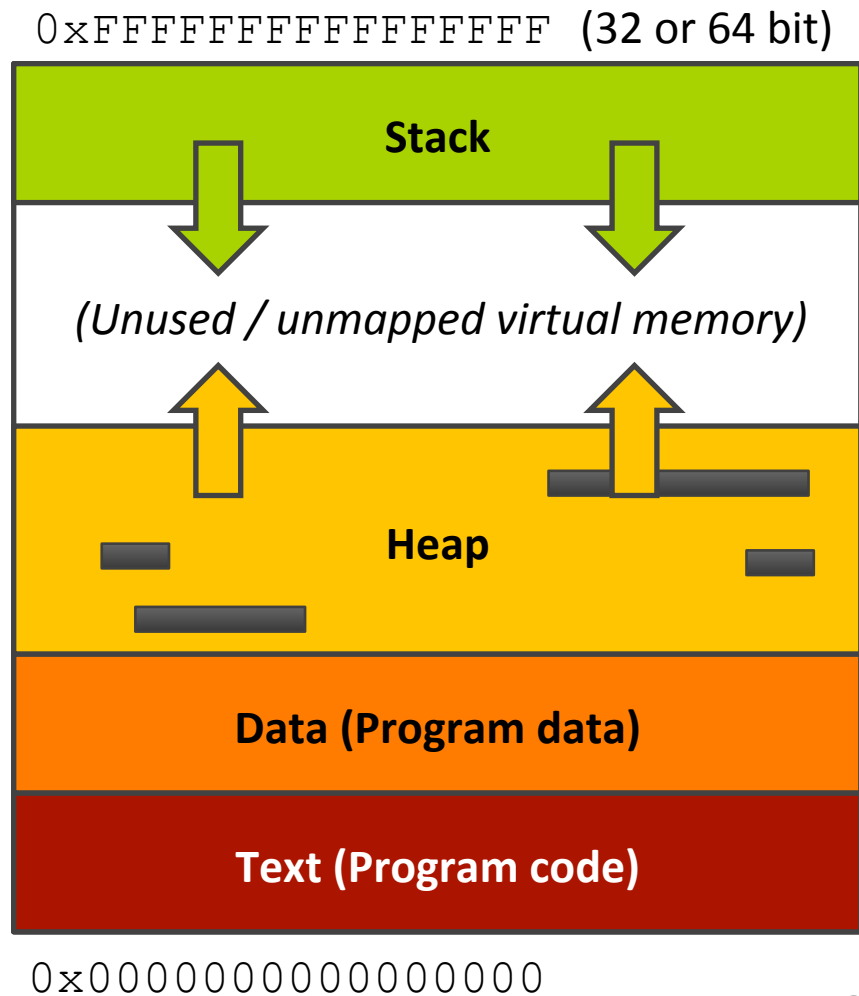
# Memory Management

- Original heap space eventually fills up
- `malloc()` requests additional space from the kernel by using `brk()` system call



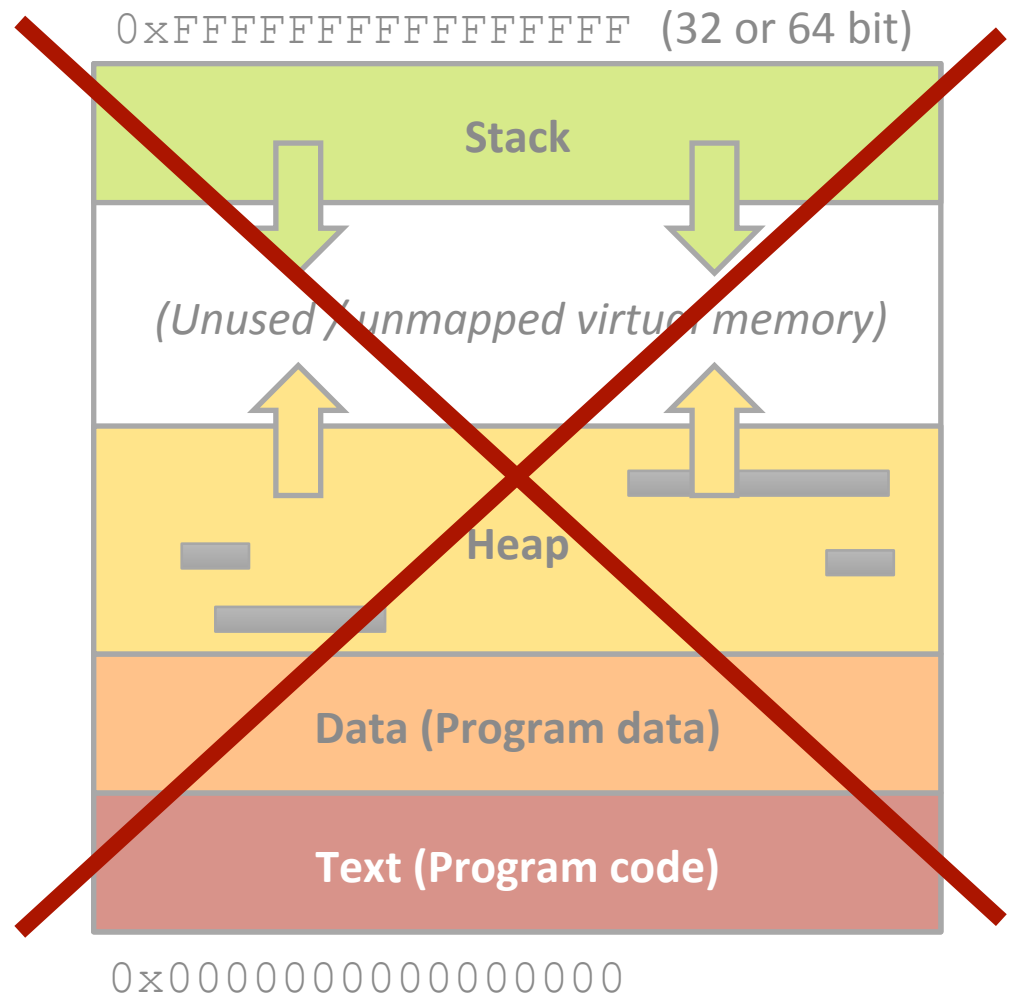
# Memory Management

➔ `free()`  
deallocates  
blocks from the  
heap



# Memory Management

- Program terminates
- OS expunges entire virtual address space
- Everything is deleted



# 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
  
- **Will learn a tool (Valgrind) in Lab 5 to analyze your program and detect / trace errors**



# 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>

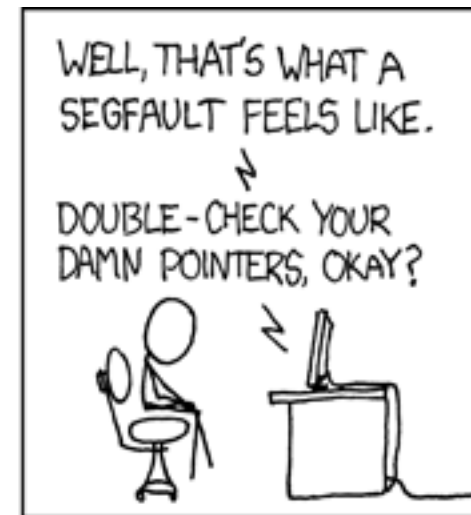
# What's the Error?

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

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



AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?



<http://xkcd.com/371/>

# Memory Management

- **What's a NULL pointer?**
  - Pointer value is 0x00000000
  - *Meaning* is that the pointer is not pointing anywhere
  
- **What happens if you dereference 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)
- **“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