UNIVERSITY OF OREGON

CIS 330:

```
   __  __      _                          __   ______     __ ______
  / / / /___  (_)  __   ____ _____  ____/ /  / ____/   _/_// ____/__    __
 / / / / __ \/ / |/_/  / __ `/ __ \/ __  /  / /      _/_/ / /  __/ /___/ /_
/ /_/ / / / / />  <   / /_/ / / / / /_/ /  / /___  _/_/  / /__/_  __/_  __/
\____/_/ /_/_/_/|_|   \__,_/_/ /_/\__,_/   \____/ /_/    \____/ /_/   /_/
```

Lecture 2:
Memory in C

April 4, 2018

Hank Childs, University of Oregon

Office Hours

- Hank's OH:
  - Tues 2pm-3pm, Friday, 11am-noon
- Hank's OH Location: 301 Deschutes Hall

- Brent's Office Hours:
  - Monday: 4-5pm
  - Weds: 4-5pm (not today)
  - Thurs: 4-5pm (not today)
- TA OH Location: 100 Deschutes Hall

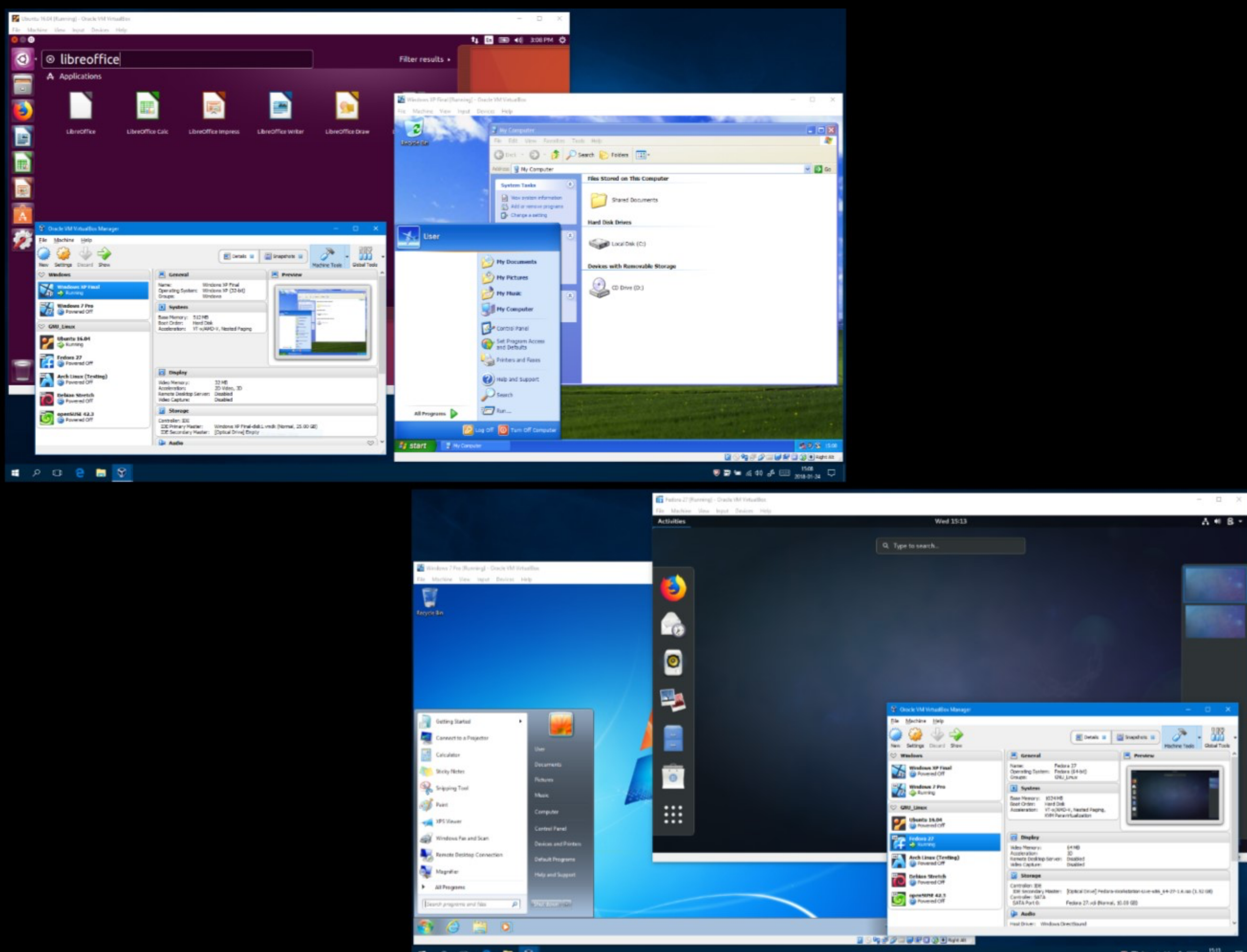
Note on Homeworks

- Project 1A: assigned Monday, due tonight
  - → means 6am Thursday
  - Will discuss again in 10 slides
- Project 2A: assigned today, due in class on Monday
- Project 1B assigned Friday, due Weds Apr 12
- Lecture this Friday (not lab)

UNIVERSITY OF OREGON

VirtualBox Demo

UNIVERSITY OF OREGON Libraries



Virtualization is useful in an incredible variety of cases, but it also involves knowing arcane details about how computers work. Additionally, it has its own unique hiccups that can be a snag for even veteran technicians. Do you know some basics about computers, and want to expand upon that through virtualizing an operating system? Show up, and we can help get you started.

WHEN

Fri, April 06

4:00 - 6:00 PM

WHERE

PSC B006 (Vislab)

https://library.uoregon.edu/scilib/psc-dearmond-makerspace

More on Piazza

Plan for today

- Quick review of Unix basics
- Project 1A
- Baby steps into C and gcc
- Memory

Plan for today

- Quick review of Unix basics
- Project 1A
- Baby steps into C and gcc
- Memory

Files

- Unix maintains a file system
  - File system controls how data is stored and retrieved
- Primary abstractions:
  - Directories
  - Files
- Files are contained within directories

Directories are hierarchical

- Directories can be placed within other directories
- “/” -- The root directory
  - Note “/”, where Windows uses “\”
- “/dir1/dir2/file1”
  - What does this mean?

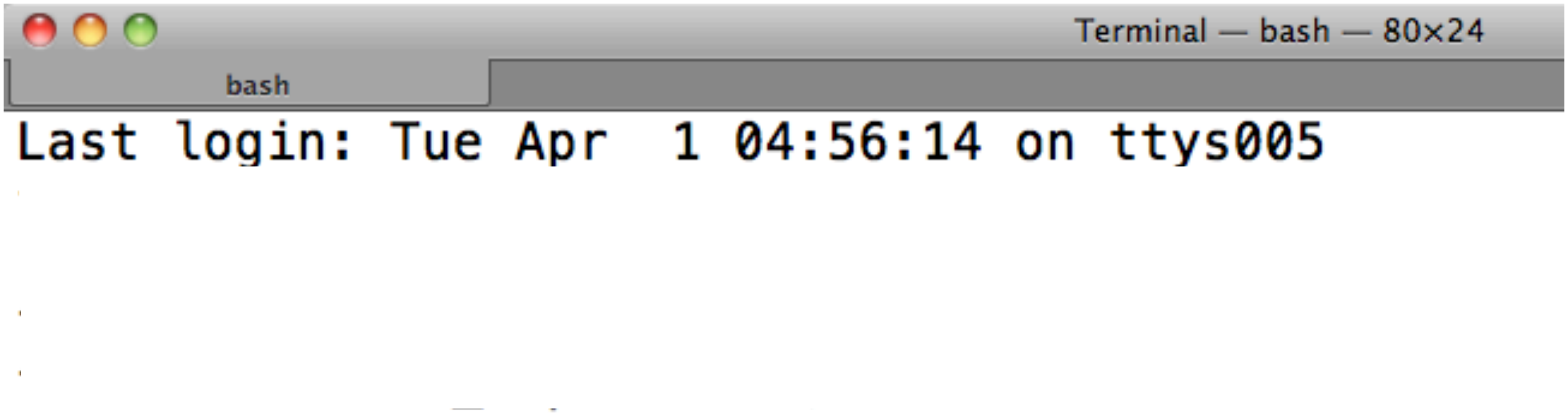
File file1 is contained in directory dir2, which is contained in directory dir1, which is in the root directory

Home directory

- Unix supports multiple users
- Each user has their own directory that they control
- Location varies over Unix implementation, but typically something like "/home/username"
- Stored in environment variables

```
fawcett:~ childs$ echo $HOME
/Users/childs
```

File manipulation



New commands: mkdir, cd, touch, ls, rmdir, rm

cd: change directory

- The shell always has a “present working directory”
  - directory that commands are relative to
- “cd” changes the present working directory
- When you start a shell, the shell is in your “home” directory

Unix commands: mkdir

- mkdir: makes a directory
  - Two flavors
    - Relative to current directory
      - mkdir dirNew
    - Relative to absolute path
      - mkdir /dir1/dir2/dirNew
        - » (dir1 and dir2 already exist)

Unix commands: rmdir

- rmdir: removes a directory
  - Two flavors
    - Relative to current directory
      - rmdir badDir
    - Relative to absolute path
      - rmdir /dir1/dir2/badDir
        - » Removes badDir, leaves dir1, dir2 in place
- Only works on empty directories!
  - "Empty" directories are directories with no files

Most Unix commands can distinguish between absolute and relative path, via the "/" at beginning of filename.
(I'm not going to point this feature out for subsequent commands.)

Unix commands: touch

- touch: "touch" a file
- Behavior:
  - If the file doesn't exist
    - → create it
  - If the file does exist
    - → update time stamp

Time stamps record the last modification to a file or directory

Will talk more about this command with build systems

Unix commands: ls

- ls: list the contents of a directory
  - Note this is "LS", not "is" with a capital 'i'
- Many flags, which we will discuss later
  - A flag is a mechanism for modifying a Unix programs behavior.
  - Convention of using hyphens to signify special status
- "ls" is also useful with "wild cards", which we will also discuss later

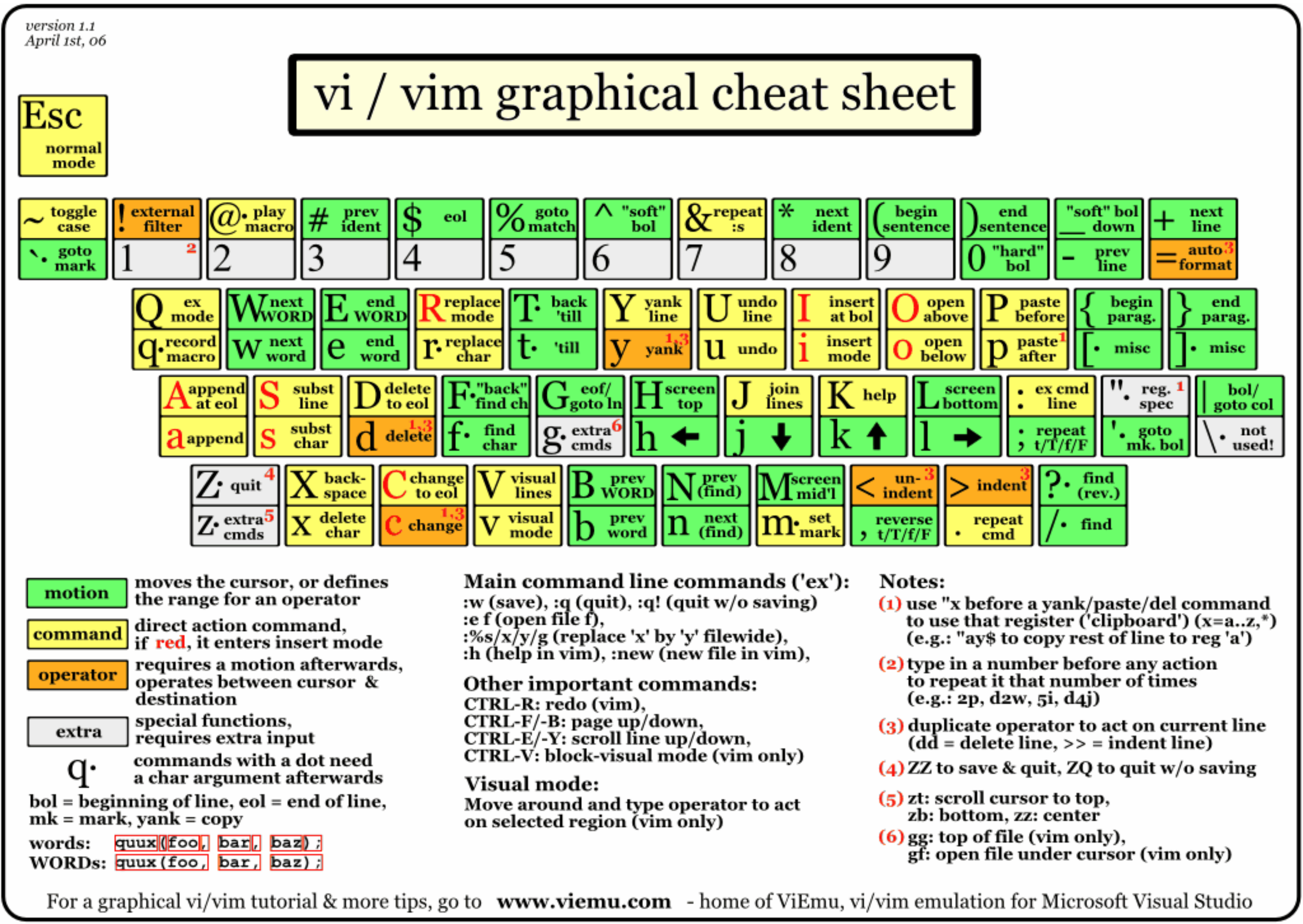
Important: “man”

- Get a man page:
- → “man rmdir” gives:

```
RMDIR(1)                  BSD General Commands Manual                  RMDIR(1)

NAME
     rmdir -- remove directories

SYNOPSIS
     rmdir [-p] directory ...

DESCRIPTION
     The rmdir utility removes the directory entry specified by each directory
     argument, provided it is empty.

     Arguments are processed in the order given.  In order to remove both a
     parent directory and a subdirectory of that parent, the subdirectory must
     be specified first so the parent directory is empty when rmdir tries to
     remove it.

     The following option is available:

     -p      Each directory argument is treated as a pathname of which all
             components will be removed, if they are empty, starting with the
             last most component.  (See rm(1) for fully non-discriminant
:
```

File Editors

- vimtutor a great start for learning “vi”
- But ask me for tips any time you see me editing



http://www.viemu.com/vi-vim-cheat-sheet.gif

Plan for today

- Quick review of Unix basics
- Project 1A
- Baby steps into C and gcc
- Memory

Project 1A

- Practice using an editor
- Must be written using editor on Unix platform
  - – I realize this is unenforceable.
  - – If you want to do it with another mechanism, I can't stop you
    - But realize this project is simply to prepare you for later projects

Project 1A

- Write >=300 words using editor (vi, emacs, other)
- Topic: what you know about C programming language
- Can’t write 300 words?
  - Bonus topic: what you want from this course
- How will you know if it is 300 words?
  - Unix command: “wc” (word count)

Unix command: wc (word count)

```
fawcett:~ childs$ vi hanks_essay
fawcett:~ childs$ wc -w hanks_essay
     252 hanks_essay
fawcett:~ childs$ wc hanks_essay
      63     252    1071 hanks_essay
fawcett:~ childs$
```

(63 = lines, 252 = words, 1071 = character)

Project 1A

CIS 330: Project #1A
Assigned: April 2nd, 2018
Due April 4, 2018
(which means submitted by 6am on April 5th, 2018)
Worth 1% of your grade

Assignment:

1) On a Unix platform (including Mac), use an editor (vi, emacs, other) to write a 300 word "essay"
    a. The purpose of the essay is to practice using an editor.
        i. Grammar will not be graded
    b. I would like to learn more about what you know about C and want from this class … I recommend you each write about that.
    c. If you run out of things to say, you don't have to write original words (do a copy/paste using vi commands: yyp)

Do not write this in another editor and copy into vi.

Also, do not put more than 100 characters onto any given line. (I want you to practice having multiple lines and navigating.) If you have more than 100 characters per line, you will receive half credit.

How to submit

- Canvas
- If you run into trouble:
  - Email me your solution

Plan for today

- Quick review of Unix basics
- Project 1A
- Baby steps into C and gcc
- Memory

GNU Compilers

- GNU compilers: open source
  - gcc: GNU compiler for C
  - g++: GNU compiler for C++

Our first gcc program

```
C02LN00GFD58:CIS330 hank$ cat t.c
#include <stdio.h>
int main()
{
   printf("hello world!\n");
}
C02LN00GFD58:CIS330 hank$ gcc t.c
C02LN00GFD58:CIS330 hank$ ./a.out
hello world!
C02LN00GFD58:CIS330 hank$
```

Unix command that prints contents of a file

Invoke gcc compiler

Name of file to compile

Default name for output programs

You should use this for Proj 2A.

Plan for today

- Quick review of Unix basics
- Project 1A
- Baby steps into C and gcc
- Memory

Reading

- 4.1 (but NOT 4.1.2 … covered later)
- 4.2
- 4.3-4.5.2 (what I assume you know from 314)
  - NOT 4.5.3 to 4.5.8
- 4.6: today's lecture

Why C?

- You can control the memory
- That helps get good performance

- If you don’t control the memory (like in other programming languages), you are likely to get poor performance

- … so let’s talk about memory

Motivation: Project 2A

Assignment: fill out this worksheet.

Location	0x8000	0x8004	0x8008	0x800c	0x8010	0x8014	0x8018
Value	0	1	1	2	3	5	8
Location	0x801c	0x8020	0x8024	0x8028	0x802c	0x8030	0x8034
Value	13	21	34	55	89	144	233
Location	0x8038	0x803c	0x8040	0x8044	0x8048	0x804c	0x8050
Value	377	610	987	1597	2584	4181	6765

Code:

```
int *A = 0x8000;
int *B[3] = { A, A+7, A+14 };
```

Note: "NOT ENOUGH INFO" is a valid answer.

Variable	Your Answer	Variable	Your Answer
A	0x8000	(A+6)-(A+3)	
&A	NOT ENOUGH INFO	*(A+6)-*(A+4)	
A[2]	1	A[5]-*(A+4)	
*A		(A+6)-B[0]	

Important Context

- Different types have different sizes:
  - int: 4 bytes
  - float: 4 bytes
  - double: 8 bytes
  - char: 1 byte
  - unsigned char: 1 byte

Important Memory Concepts in C (1/9): Stack versus Heap

- You can allocate variables that only live for the invocation of your function
  - Called stack variables (will talk more about this later)
- You can allocated variables that live for the whole program (or until you delete them)
  - Called heap variables (will talk more about this later as well)

Important Memory Concepts in C (2/9): Pointers

- Pointer: points to memory location
  - Denoted with ‘*’
  - Example: “int *p”
    - pointer to an integer
  - You need pointers to get to heap memory
- Address of: gets the address of memory
  - Operator: ‘&’
  - Example:

```
int x;
int *y = &x;
```

Important Memory Concepts in C (3/9): Memory allocation

- Special built-in function to allocate memory from heap: malloc
  - Interacts with Operating System
  - Argument for malloc is how many bytes you want
- Also built-in function to deallocate memory: free

free/malloc example

Enables compiler to see functions that aren't in this file. More on this next week.

sizeof is a built in function in C. It returns the number of bytes for a type (4 bytes for int).

```
#include <stdlib.h>
int main()
{
    /* allocates memory */
    int *ptr = malloc(2*sizeof(int));

    /* deallocates memory */
    free(ptr);
}
```

don't have to say how many bytes to free … the OS knows

Important Memory Concepts in C (4/9): Arrays

- Arrays lie in contiguous memory
  - So if you know address to one element, you know address of the rest
- int *a = malloc(sizeof(int)*1);
  - a single integer
  - … or an array of a single integer
- int *a = malloc(sizeof(int)*2);
  - an array of two integers
  - first integer is at ‘a’
  - second integer is at the address ‘a+4’
    - Tricky point here, since C/C++ will refer to it as ‘a+1’

Important Memory Concepts in C (5/9): Dereferencing

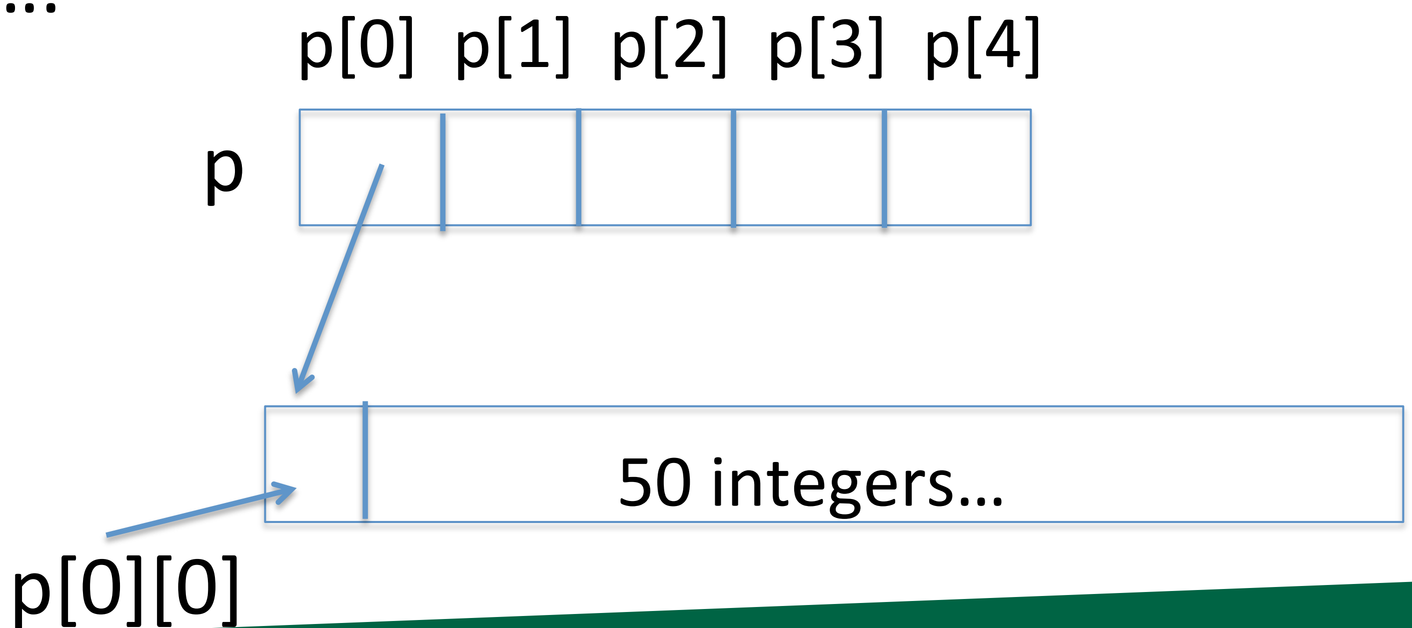
- There are two operators for getting the value at a memory location: *, and []
  - This is called deferencing
    - * = "dereference operator"
- int *p = malloc(sizeof(int)*1);
- *p = 2; /* sets memory p points to to have value 2 */
- p[0] = 2; /* sets memory p points to to have value 2 */

Important Memory Concepts in C (6/9): pointer arithmetic

- int *p = malloc(sizeof(int)*5);
- C/C++ allows you to modify pointer with math operations
  - called pointer arithmetic
  - "does the right thing" with respect to type
    - int *p = malloc(sizeof(int)*5);
    - p+1 is 4 bytes bigger than p!!
- Then:
  - "p+3" is the same as "&(p[3])" (ADDRESSES)
  - "*(p+3)" is the same as "p[3]" (VALUES)

Important Memory Concepts in C (7/9)
Pointers to pointers

- int **p = malloc(sizeof(int *)*5);
- p[0] = malloc(sizeof(int)*50);
- ….



Important Memory Concepts in C (8/9): Hexadecimal address

- Addresses are in hexadecimal
- int *A = 0x8000;
- Then A+1 is 0x8004. (Since int is 4 bytes)

Important Memory Concepts in C (9/9) NULL pointer

- int *p = NULL;
- often stored as address 0x0000000
- used to initialize something to a known value
  - And also indicate that it is uninitialized…

Project 2A

- You now know what you need to do Project 2A
  - But: practice writing C programs and testing yourself!!
  - Hint: you can printf with a pointer

```
fawcett:VIS2016 childs$ cat t.c
#include <stdlib.h>
#include <stdio.h>
int main()
{
    /* allocates memory */
    int *ptr = malloc(2*sizeof(int));
    printf("%p\n", ptr);
}
fawcett:VIS2016 childs$ gcc t.c
fawcett:VIS2016 childs$ ./a.out
0x100100080
```

Project 2A

- Assigned now
- Worksheet. You print it out, complete it on your own, and bring it to class.
- Due Monday 10am in class
  - Graded in class
- No Piazza posts on this please
- Practice with C, vi, gcc, printf

Memory Segments

- Von Neumann architecture: one memory space, for both instructions and data
- → so break memory into “segments”
  - … creates boundaries to prevent confusion
- 4 segments:
  - Code segment
  - Data segment
  - Stack segment
  - Heap segment

Code Segment

- Contains assembly code instructions
- Also called text segment
- This segment is modify-able, but that's a bad idea
  - "Self-modifying code"
    - Typically ends in a bad state very quickly.

Data Segment

- Contains data not associated with heap or stack
  - global variables
  - statics (to be discussed later)
  - character strings you've compiled in

```
char *str = "hello world\n"
```

Stack: data structure for collection

- A stack contains things
- It has only two methods: push and pop
  - Push puts something onto the stack
  - Pop returns the most recently pushed item (and removes that item from the stack)
- LIFO: last in, first out

Imagine a stack of trays.
You can place on top (push).
Or take one off the top (pop).

Stack

- Stack: memory set aside as scratch space for program execution
- When a function has local variables, it uses this memory.
  - When you exit the function, the memory is lost

Stack

- The stack grows as you enter functions, and shrinks as you exit functions.
  - This can be done on a per variable basis, but the compiler typically does a grouping.
    - Some exceptions (discussed later)
- Don't have to manage memory: allocated and freed automatically

Heap

- Heap (data structure): tree-based data structure
- Heap (memory): area of computer memory that requires explicit management (malloc, free).
- Memory from the heap is accessible any time, by any function.
  - Contrasts with the stack

Memory Segments

```
+------------+
|            |
|    text    |  (fixed size)
|            |
+------------+
|            |
|    data    |  (fixed size)
|            |
+------------+
|    stack   |  | growth
+------------+  V
|            |
|    free    |
|            |
+------------+  ^
|    heap    |  | growth
+------------+
```

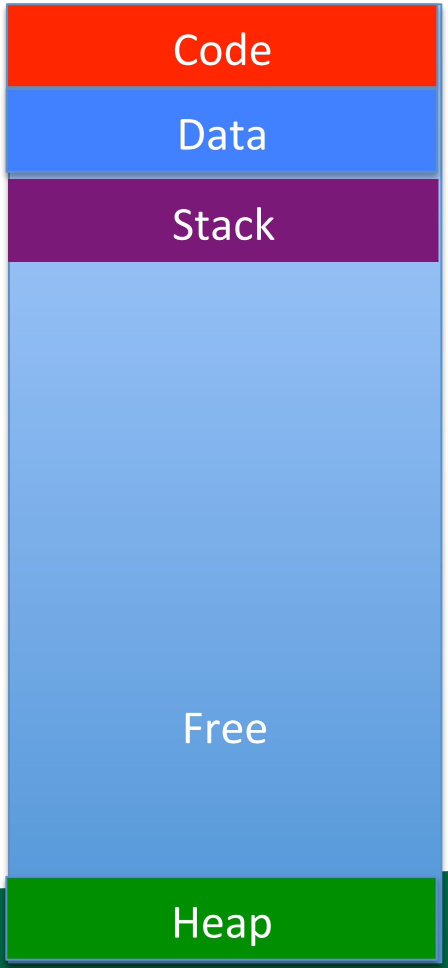
Source: http://www.cs.uwm.edu/classes/cs315/Bacon/

Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/ Deallocation	Automatic	Explicit

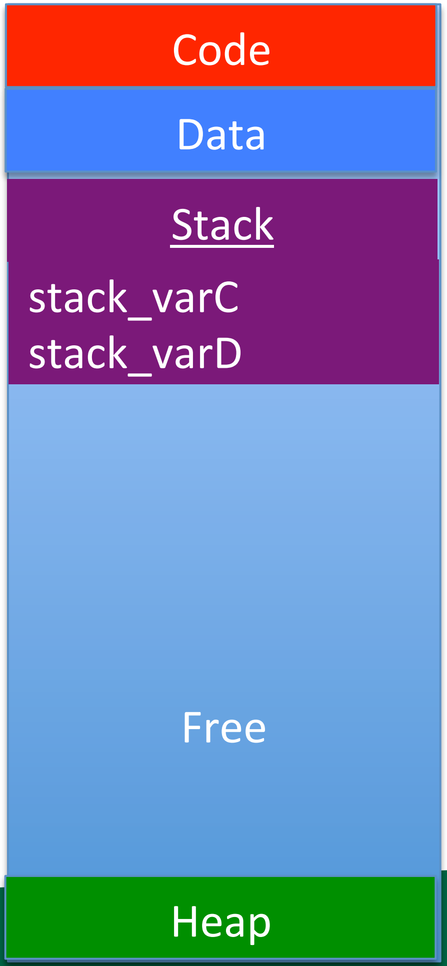
How stack memory is allocated into Stack Memory Segment

```
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```



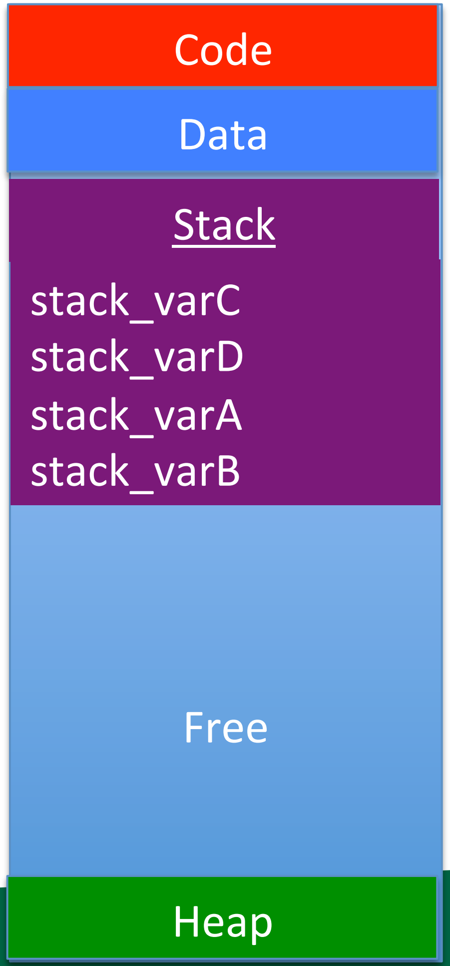
How stack memory is allocated into Stack Memory Segment

```
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```



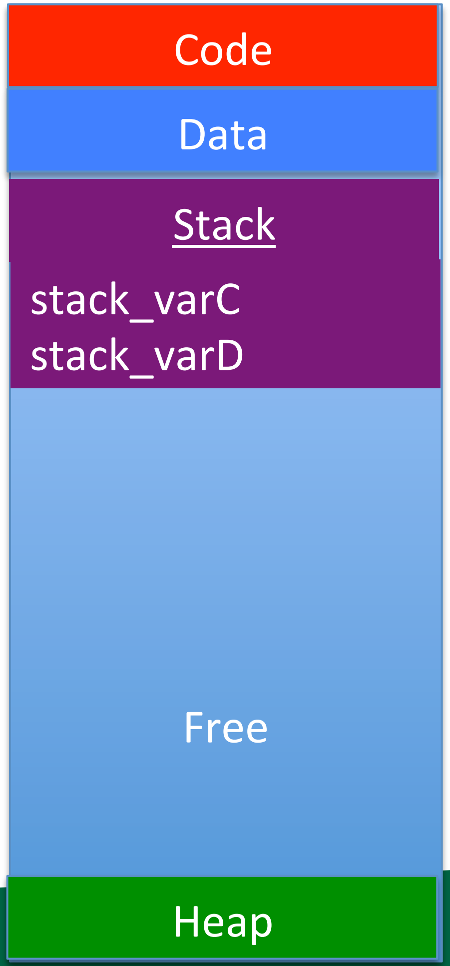
How stack memory is allocated into Stack Memory Segment

```
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```



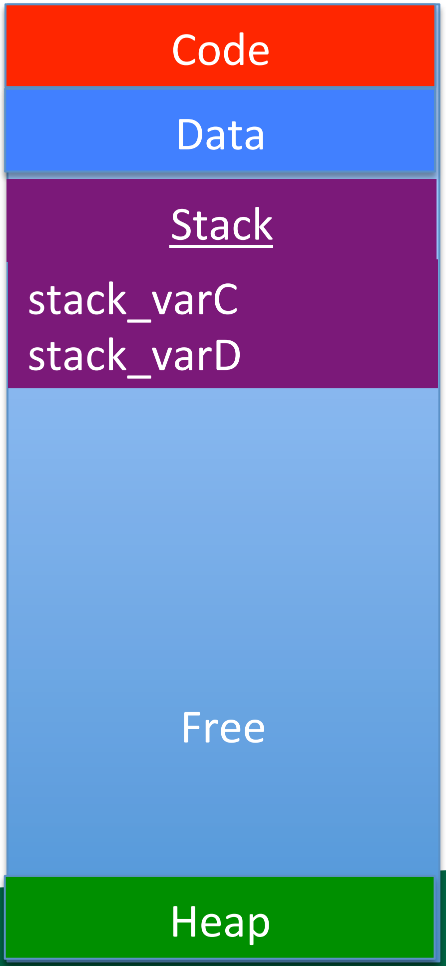
How stack memory is allocated into Stack Memory Segment

```
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```



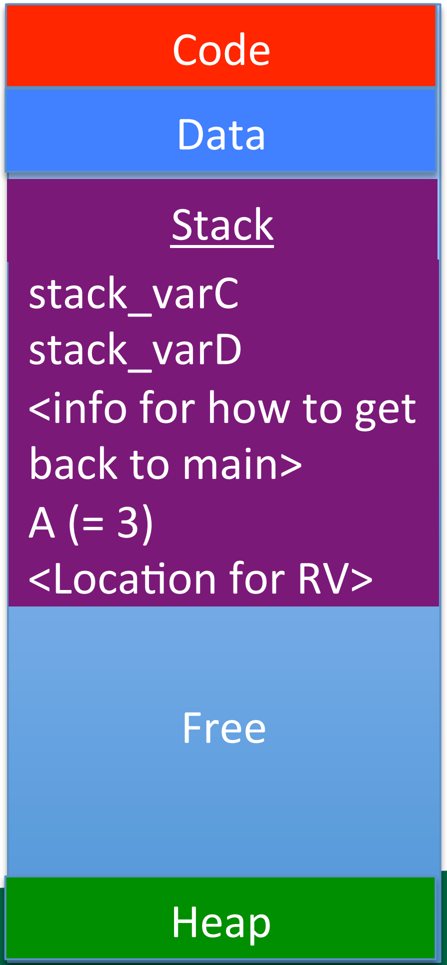
How stack memory is allocated into Stack Memory Segment

```
int doubler(int A)
{
   int stack_varA;
   stack_varA = 2*A;
   return stack_varA;
}

int main()
{
   int stack_varC;
   int stack_varD = 3;
   stack_varC = doubler(stack_varD);
}
```



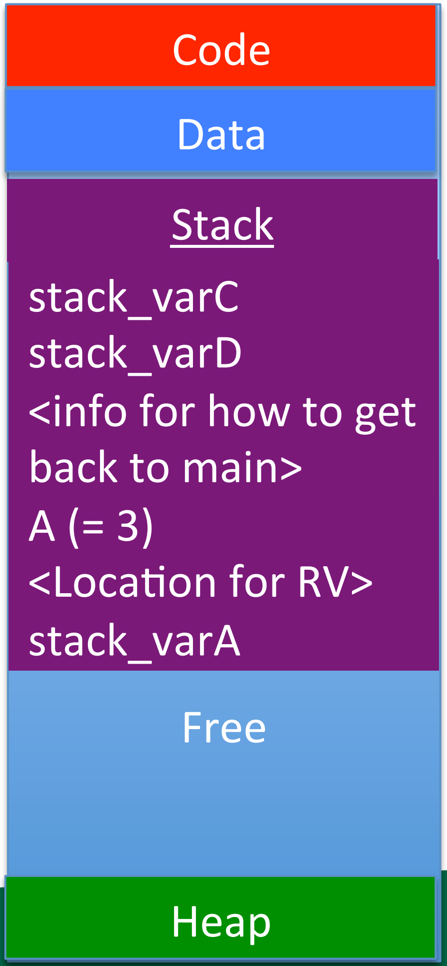
How stack memory is allocated into Stack Memory Segment

```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```



How stack memory is allocated into Stack Memory Segment

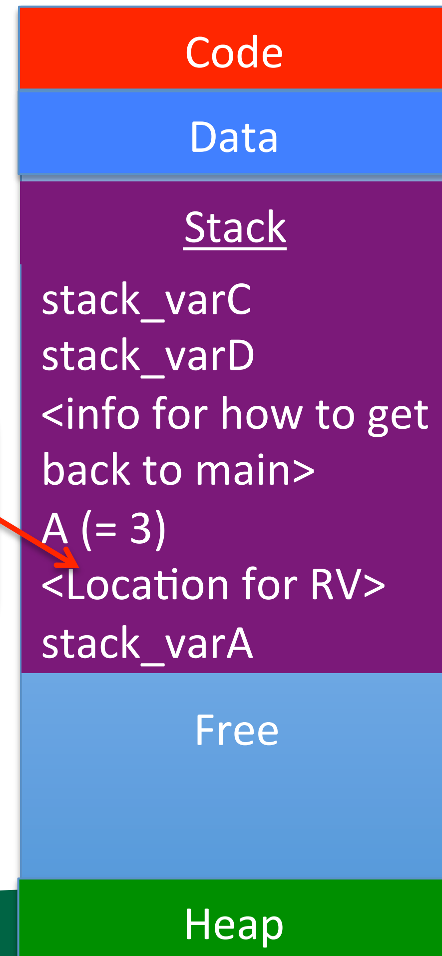
```
int doubler(int A)
{
   int stack_varA;
   stack_varA = 2*A;
   return stack_varA;
}

int main()
{
   int stack_varC;
   int stack_varD = 3;
   stack_varC = doubler(stack_varD);
}
```



How stack memory is allocated into Stack Memory Segment

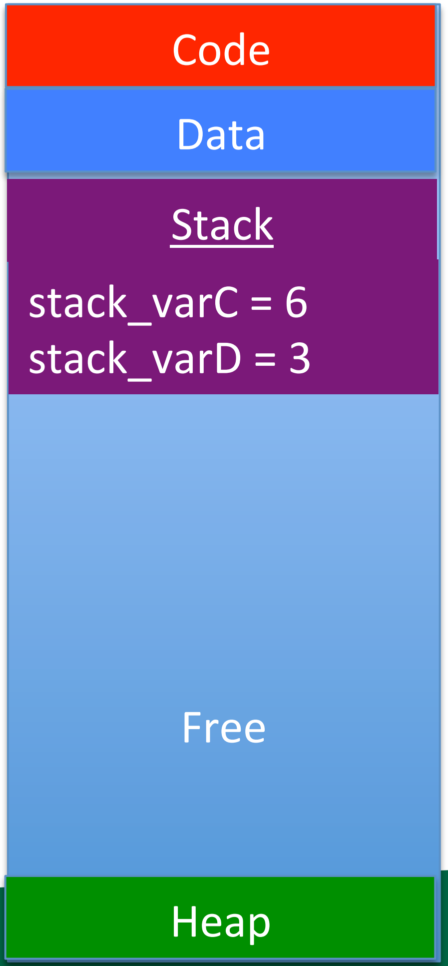
```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```

Return copies into location specified by calling function



How stack memory is allocated into Stack Memory Segment

```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```



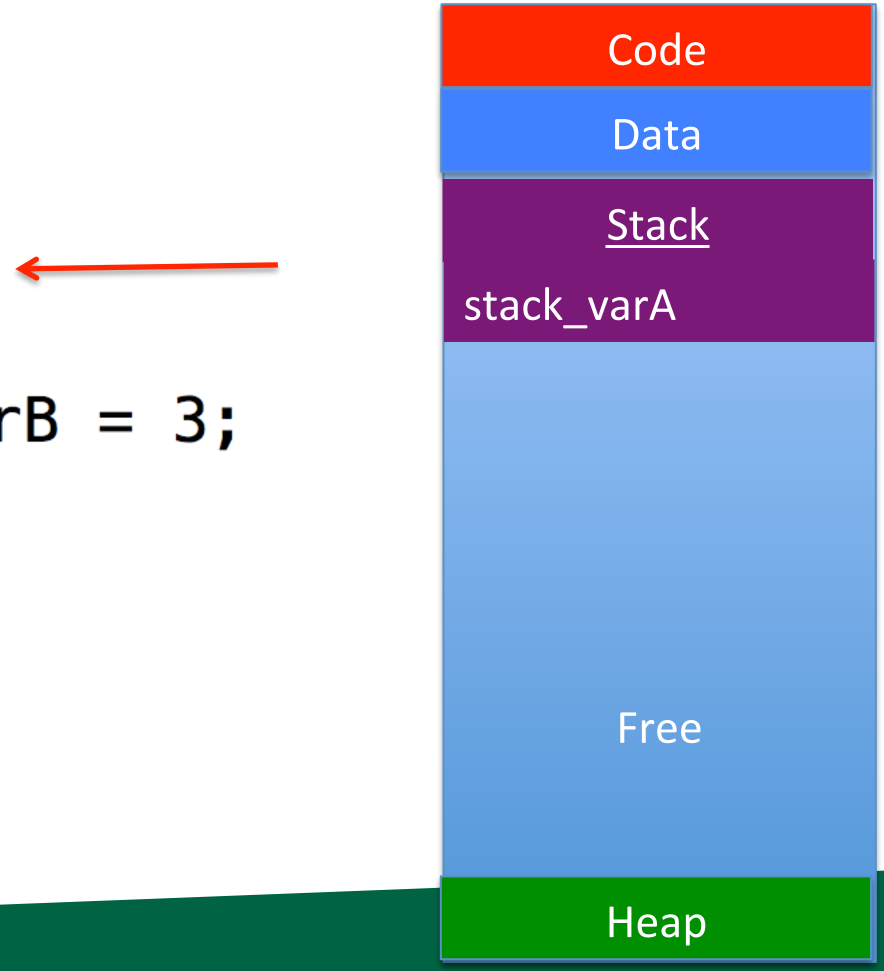
This code is very problematic … why?

```
int *foo()
{
    int stack_varC[2] = { 0, 1 };
    return stack_varC;
}
int *bar()
{
    int stack_varD[2] = { 2, 3 };
    return stack_varD;
}

int main()
{
    int *stack_varA, *stack_varB;
    stack_varA = foo();
    stack_varB = bar();
    stack_varA[0] *= stack_varB[0];
}
```

foo and bar are returning addresses that are on the stack … they could easily be overwritten (and bar's stack_varD overwrites foo's stack_varC in this program)

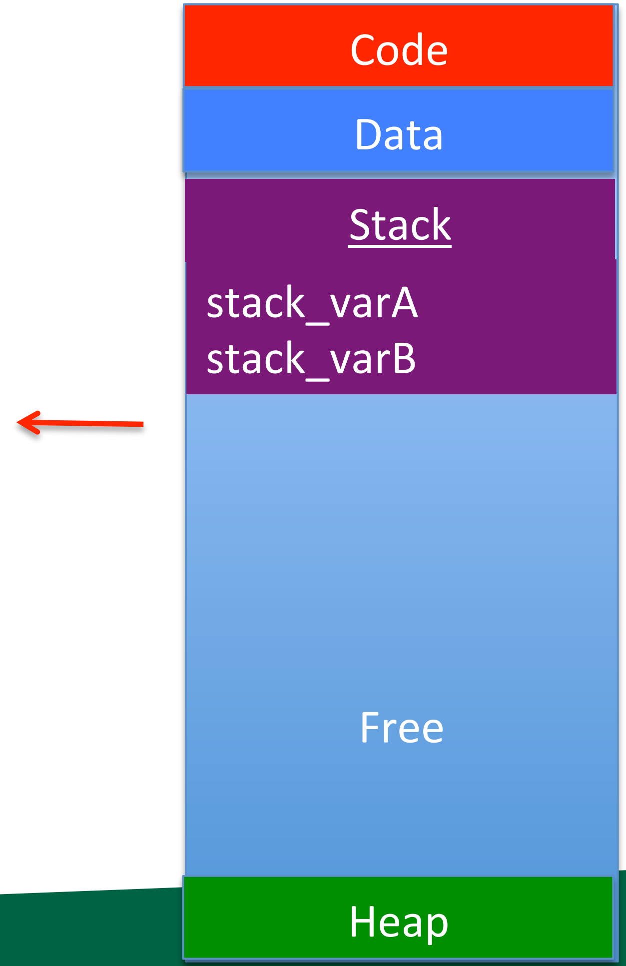
Nested Scope

```
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```



Nested Scope

```
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```



Nested Scope

```
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```

You can create new scope within a function by adding '{' and '}'.

Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/ Deallocation	Automatic	Explicit
Access	Fast	Slower

Memory pages associated with stack are almost always immediately available.

Memory pages associated with heap may be located anywhere ... may be caching effects

Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/ Deallocation	Automatic	Explicit
Access	Fast	Slower
Variable scope	Limited	Unlimited

Variable scope: stac

```
int *foo()
{
    int stack_varA[2] = { 0, 1 };
    return stack_varA;
}

int *bar()
{
    int *heap_varB;
    heap_varB = malloc(sizeof(int)*2);
    heap_varB[0] = 2;
    heap_varB[1] = 2;
    return heap_varB;
}

int main()
{
    int *stack_varA;
    int *stack_varB;
    stack_varA = foo(); /* problem */
    stack_varB = bar(); /* still good */
}
```

foo is bad code … never return memory on the stack from a function

bar returned memory from heap

The calling function – i.e., the function that calls bar – must understand this and take responsibility for calling free.

If it doesn't, then this is a "memory leak".

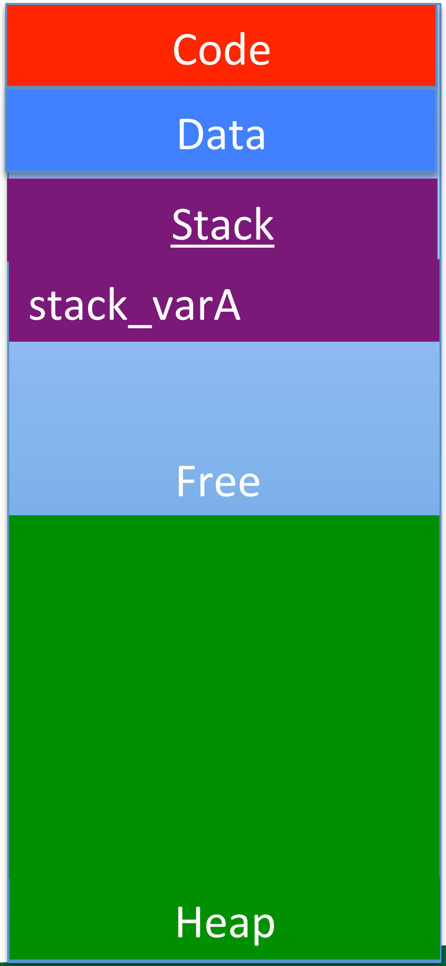
Memory leaks

It is OK that we are using the heap … that’s what it is there for

The problem is that we lost the references to the first 49 allocations on heap

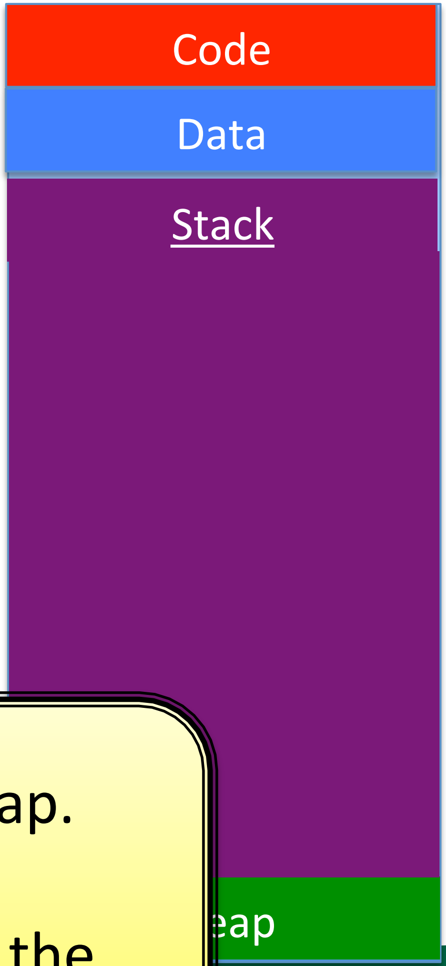
The heap’s memory manager will not be able to re-claim them … we have effectively limited the memory available to the program.

```
{
    int i;
    int stack_varA;
    for (i = 0 ; i < 50 ; i++)
       stack_varA = bar();
}
```



Running out of memory (stack)

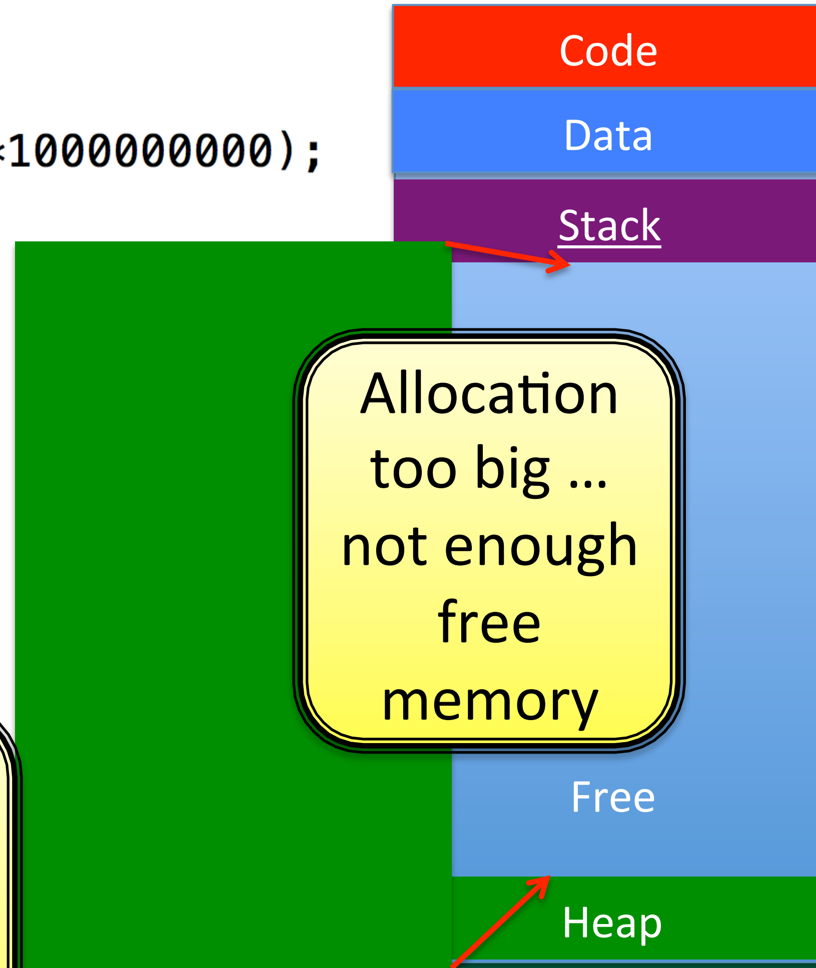
```
int endless_fun()
{
    endless_fun();
}

int main()
{
    endless_fun();
}
```



stack overflow: when the stack runs into the heap.
<u>There is no protection for stack overflows</u>.
(Checking for it would require coordination with the heap's memory manager on every function calls.)

Running out of memory (heap)

```
int *heaps_o_fun()
{
    int *heap_A = malloc(sizeof(int)*1000000000);
    return heap_A;
}

int main()
{
    int *stack_A;
    stack_A = heaps_o_fun();
}
```



Allocation too big … not enough free memory

If the heap memory manager doesn't have room to make an allocation, then malloc returns NULL …. a more graceful error scenario.

Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/ Deallocation	Automatic	Explicit
Access	Fast	Slower
Variable scope	Limited	Unlimited
Fragmentation	No	Yes

Memory Fragmentation

- Memory fragmentation: the memory allocated on the heap is spread out of the memory space, rather than being concentrated in a certain address space.

Memory Fragmentation

```
int *bar()
{
   int *heap_varA;
   heap_varA = malloc(sizeof(int)*2);
   heap_varA[0] = 2;
   heap_varA[1] = 2;
   return heap_varA;
}

int main()
{
   int i;
   int stack_varA[50];
   for (i = 0 ; i < 50 ; i++)
      stack_varA[i] = bar();
   for (i = 0 ; i < 25 ; i++)
      free(stack_varA[i*2]);
}
```

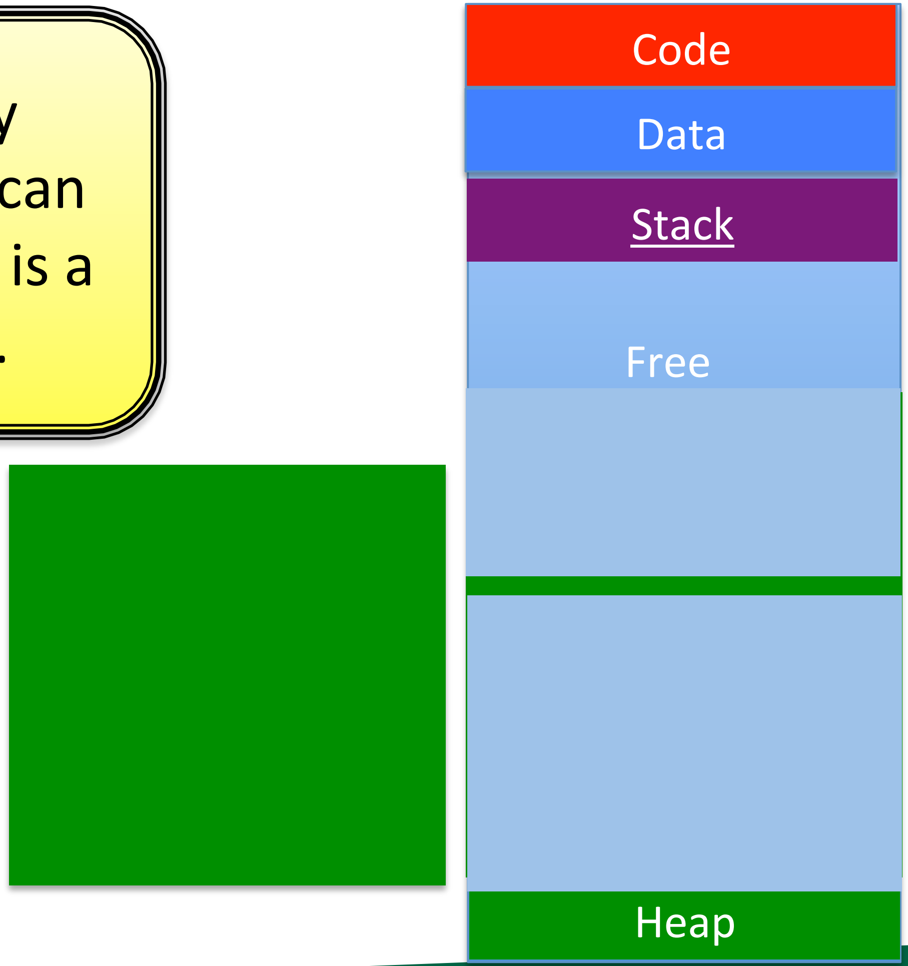
Negative aspects of <u>fragmentation?</u>

(1) can't make big allocations

(2) losing cache coherency

Fragmentation and Big Allocations

Even if there is lots of memory available, the memory manager can only accept your request if there is a big enough contiguous chunk.



Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/ Deallocation	Automatic	Explicit
Access	Fast	Slower
Variable scope	Limited	Unlimited
Fragmentation	No	Yes

Memory Errors

- Array bounds read

```
int main()
{
    int var;
    int arr[3] = { 0, 1, 2 };
    var=arr[3];
}
```

- Array bounds write

```
int main()
{
    int var = 2;
    int arr[3];
    arr[3]=var;
}
```

Memory Errors

- Free memory read / free memory write

```
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    var[0] = var[1];
}
```

When does this happen in real-world scenarios?

Memory Errors

- Freeing unallocated memory

```
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    free(var);
}
```

When does this happen in real-world scenarios?

Vocabulary: “dangling pointer”: pointer that points to memory that has already been freed.

Memory Errors

- Freeing non-heap memory

```
int main()
{
    int var[2]
    var[0] = 0;
    var[1] = 2;
    free(var);
}
```

When does this happen in real-world scenarios?

Memory Errors

- NULL pointer read / write

```
int main()
{
    char *str = NULL;
    printf(str);
    str[0] = 'H';
}
```

- NULL is never a valid location to read from or write to, and accessing them results in a "segmentation fault"
  - …. remember those memory segments?

When does this happen in real-world scenarios?

Memory Errors

- Unitialized memory read

```
int main()
{
    int *arr = malloc(sizeof(int)*10);
    int V2=arr[3];
}
```

When does this happen in real-world scenarios?