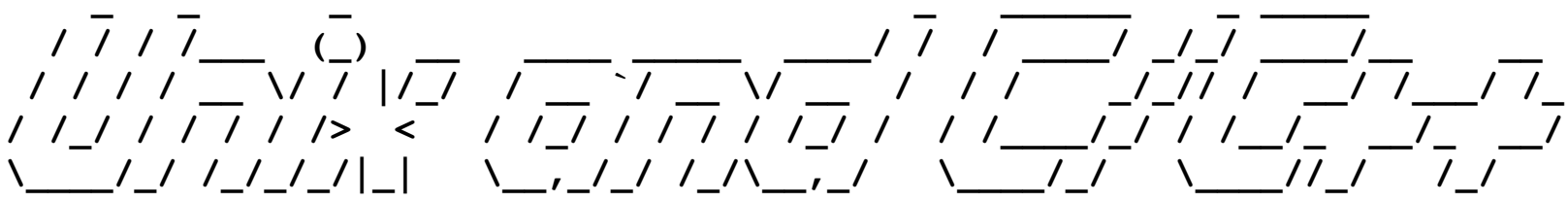
UNIVERSITY OF OREGON

CIS 330:

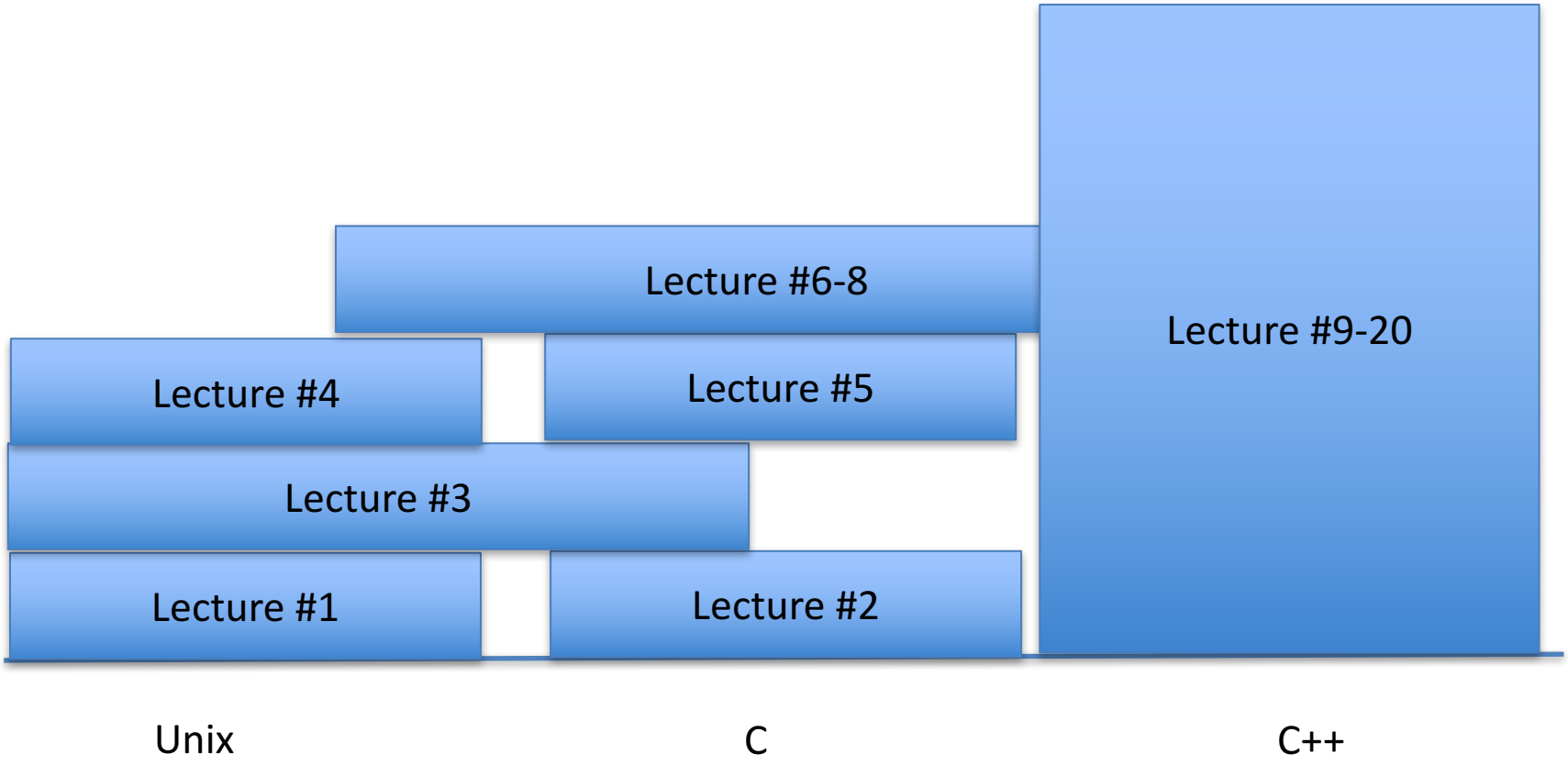


Lecture 3:
Permissions and More Memory Stuff

April 6th, 2018

Hank Childs, University of Oregon

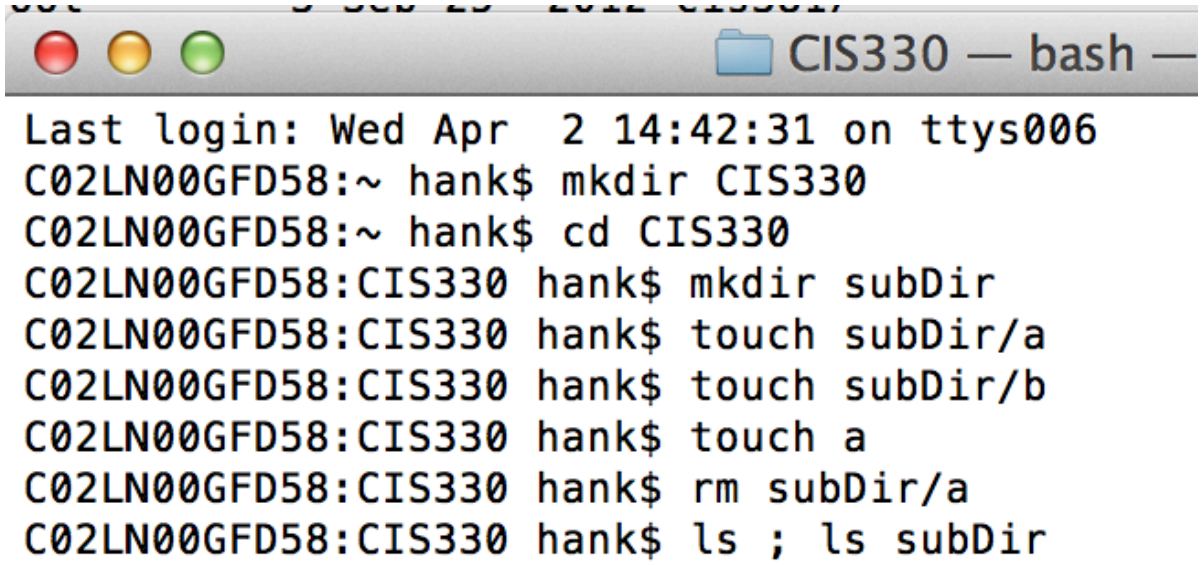
Lectures



My goal is to arrange topics such that more work can be put earlier in the course.

Quiz

- What is the output of this sequence?



Semi-colon: issue the first command, then the second right afterwards

Output from first command will be on one line, with second command on the next line.

Unix systems

- Four basic use cases
  - Personal use machines
  - Servers
  - Embedded
  - Compute clusters

Are there more?
(this is off the top of my head)

In many of these scenarios, there is a system administrator who makes an “image” of the OS that they “clone” for each machine.

I have used Unix actively since 1994, but only did system administration 2005-2009 when I had a Linux box in my home.

Outline

- Permissions
- Project 1B Overview
- More on memory / arrays / pointers

Outline

- Permissions
- Project 1B Overview
- More on memory / arrays / pointers

Permissions: System Calls

- System calls: a request from a program to the OS to do something on its behalf
  - … including accessing files and directories

- System calls:
  - Typically exposed through functions in C library
  - Unix utilities (cd, ls, touch) are programs that call these functions

Permissions in Unix are enforced via system calls.

Permissions: Unix Groups

- Groups are a mechanism for saying that a subset of Unix users are related
  - In 2014, we had a “330_S14” unix group on ix
  - Members:
    - Me
    - 2 GTFs

CIS uses “groupctl”

The commands for creating a group tend to vary, and are often done by a system administrator

Permissions

- Permissions are properties associated with files and directories
  - System calls have built-in checks to permissions
    - Only succeed if proper permissions are in place
- Three classes of permissions:
  - User: access for whoever owns the file
    - You can prevent yourself from accessing a file!
      - (But you can always change it back)
  - Group: allow a Unix group to access a file
  - Other: allow anyone on the system to access a file

Three types of permissions

- Read
- Write
- Execute (see next slide)

Executable files

- An executable file: a file that you can invoke from the command line
  - Scripts
  - Binary programs
- The concept of whether a file is executable is linked with file permissions

There are 9 file permission attributes

- Can user read?
- Can user write?
- Can user execute?
- Can group read?
- Can group write?
- Can group execute?
- Can other read?
- Can other write?
- Can other execute?

User = “owner”
Other = “not owner, not group”

A bunch of bits … we could represent this with binary

Translating R/W/E permissions to binary

#	Permission	rwx
7	full	111
6	read and write	110
5	read and execute	101
4	read only	100
3	write and execute	011
2	write only	010
1	execute only	001
0	none	000

Image from wikipedia

Which of these modes make sense? Which don't?

We can have separate values (0-7) for user, group, and other

Unix command: chmod

- chmod: change file mode

- chmod 750 <filename>
  - User gets 7 (rwx)
  - Group gets 5 (rx)
  - Other gets 0 (no access)

Lots of options to chmod
(usage shown here is most common)

Manpage for chmod

- "man chmod"

CIS330 — less — 80×24

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

NAME
     chmod -- change file modes or Access Control Lists

SYNOPSIS
     chmod [-fv] [-R [-H | -L | -P]] mode file ...
     chmod [-fv] [-R [-H | -L | -P]] [-a | +a | =a] ACE file ...
     chmod [-fhv] [-R [-H | -L | -P]] [-E] file ...
     chmod [-fhv] [-R [-H | -L | -P]] [-C] file ...
     chmod [-fhv] [-R [-H | -L | -P]] [-N] file ...

DESCRIPTION
     The chmod utility modifies the file mode bits of the listed files as
     specified by the mode operand. It may also be used to modify the Access
     Control Lists (ACLs) associated with the listed files.

     The generic options are as follows:

     -f      Do not display a diagnostic message if chmod could not modify the
             mode for file.

:
```

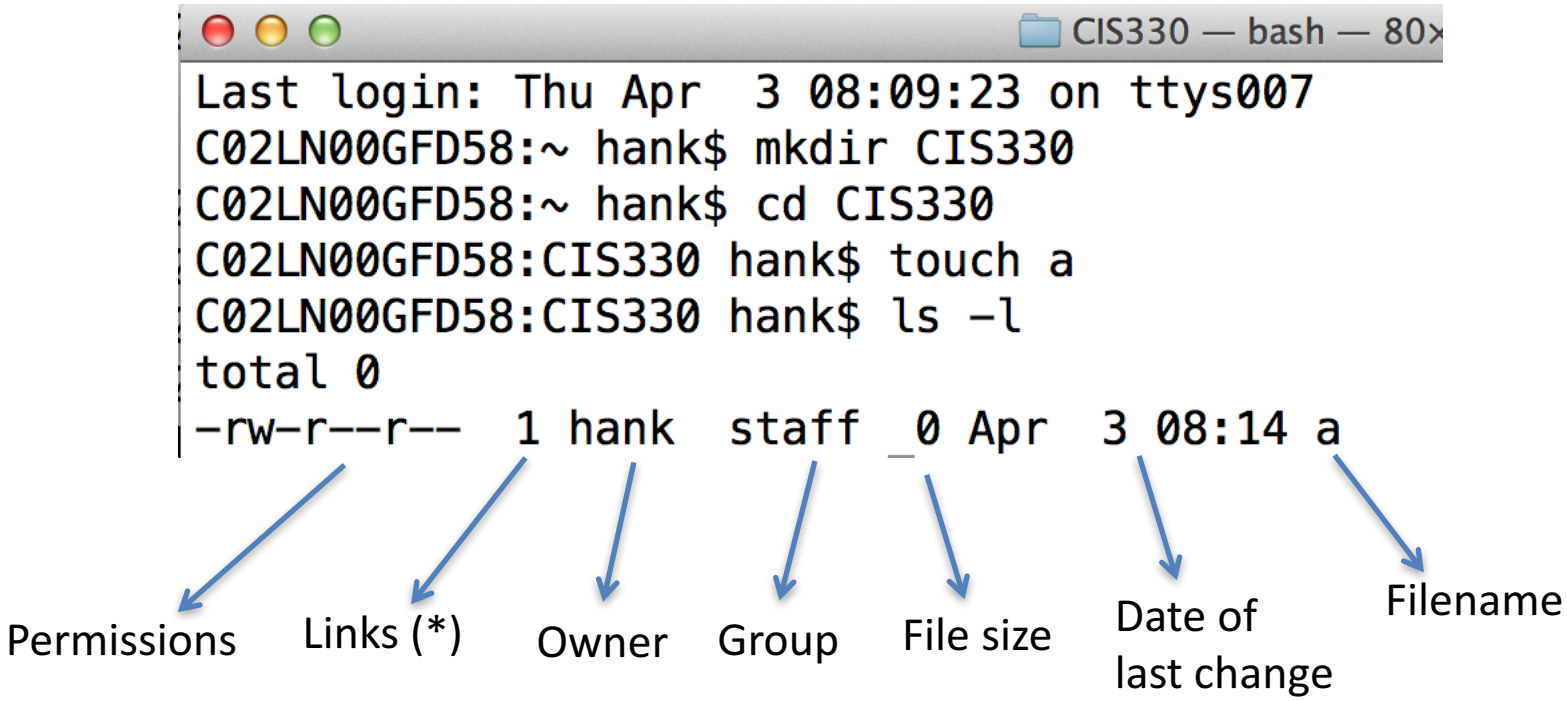
Unix commands for groups

- chgrp: changes the group for a file or directory
  - chgrp <group> <filename>

- groups: lists groups you are in

ls -l

- Long listing of files



How to interpret this?

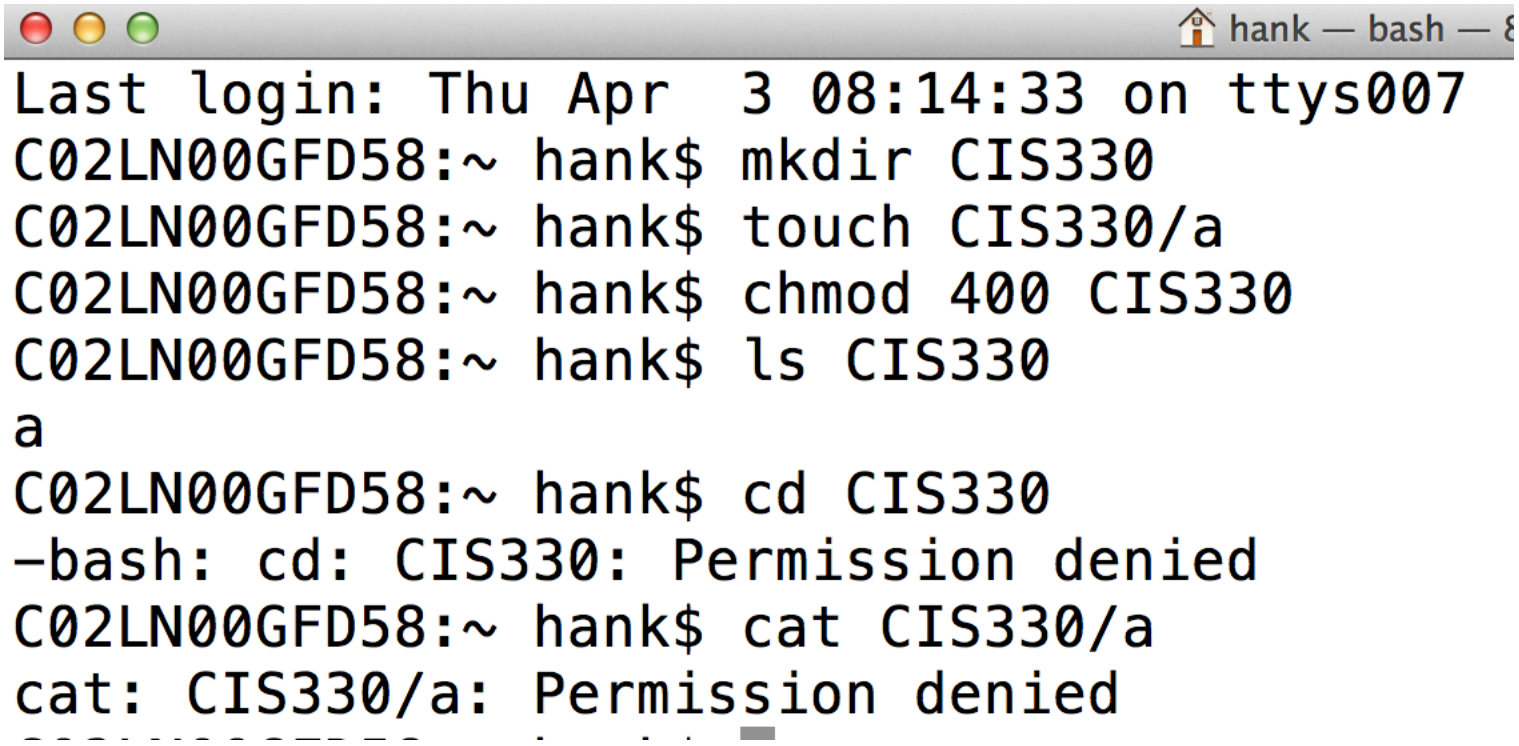
Permissions and Directories

- You can only enter a directory if you have "execute" permissions to the directory

- Quiz: a directory has permissions "400". What can you do with this directory?

Answer: it depends on what permissions a system call requires.

Directories with read, but no execute



Outline

- Permissions
- Project 1B Overview
- More on memory / arrays / pointers

Unix scripts

- Scripts
  - Use an editor (vi/emacs/other) to create a file that contains a bunch of Unix commands
  - Give the file execute permissions
  - Run it like you would any program!!

Unix scripts

- Arguments
  - Assume you have a script named “myscript”
  - If you invoke it as “myscript foo bar”
  - Then
    - $# == 2
    - $1 == foo
    - $2 == bar

Project 1B

- Summary: write a script that will create a specific directory structure, with files in the directories, and specific permissions.

Project 1B

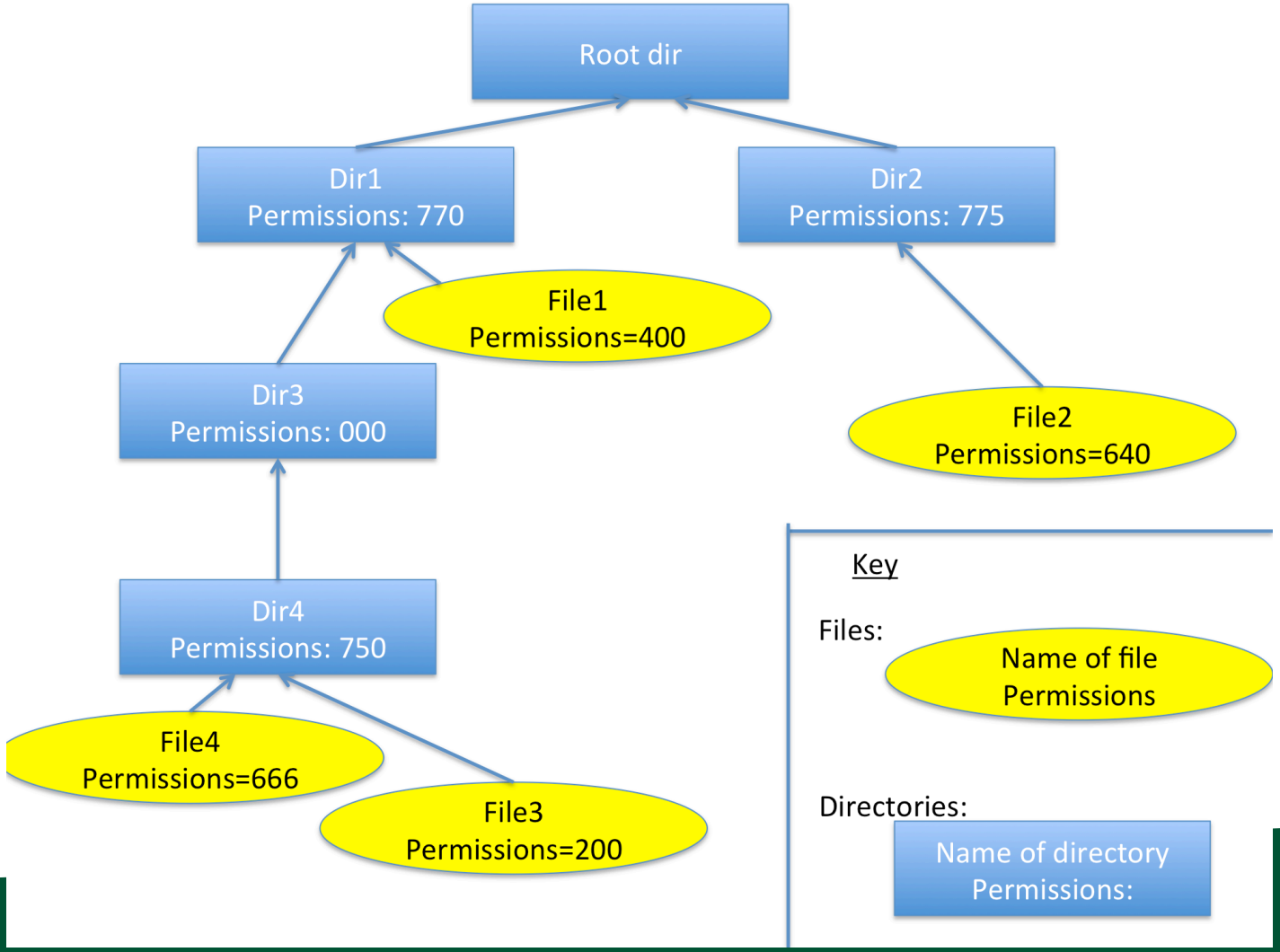
CIS 330: Project #1B
Assigned: April 6th, 2018
Due April 11th, 2018
(which means submitted by 6am on April 12th, 2018)
Worth 2% of your grade

Assignment: Create a shell script that will create a directory structure and files within that directory structure, all with the specified file permissions. The script should be named "proj1b.sh". (A consistent name will help with grading.)

Note: you are only allowed to use the following commands: mkdir, touch, cd, chmod, mv, cp, rm, rmdir. (You do not need to use all of these commands to successfully complete the assignment.)

Project 1B

The directory structure should be:



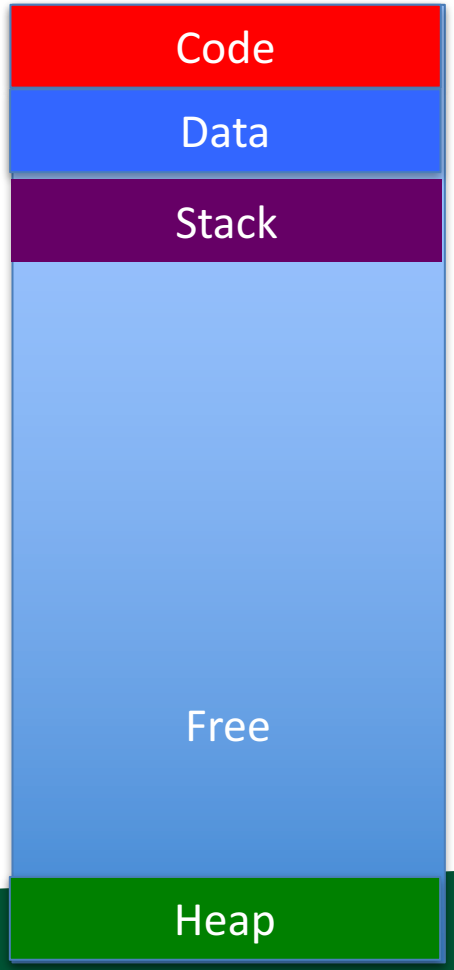
(Finish Lecture 2)

Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/Deal location	Automatic	Explicit

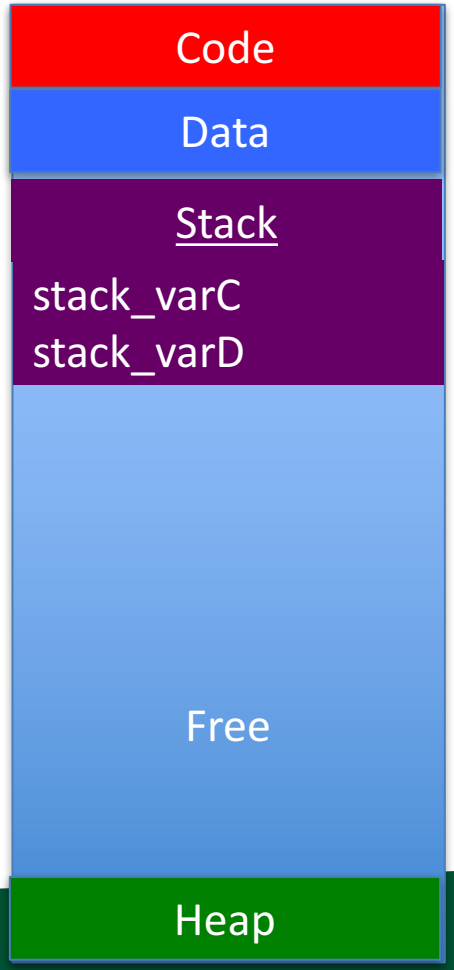
How stack memory is allocated into Stack Memory Segment

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

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



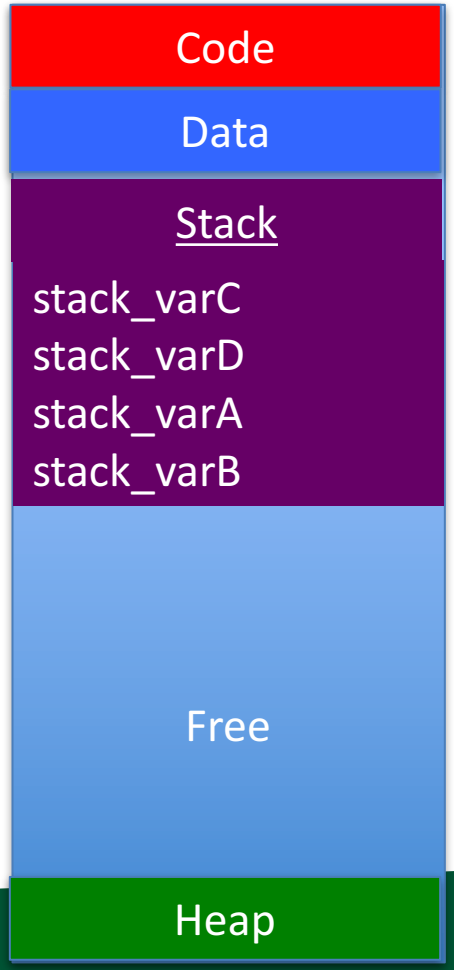
How stack memory is allocated into Stack Memory Segment

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

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



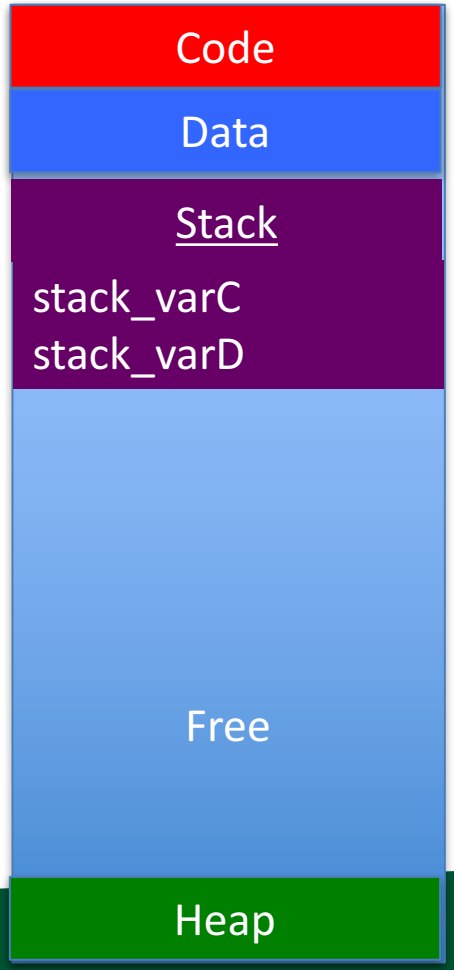
How stack memory is allocated into Stack Memory Segment

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

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



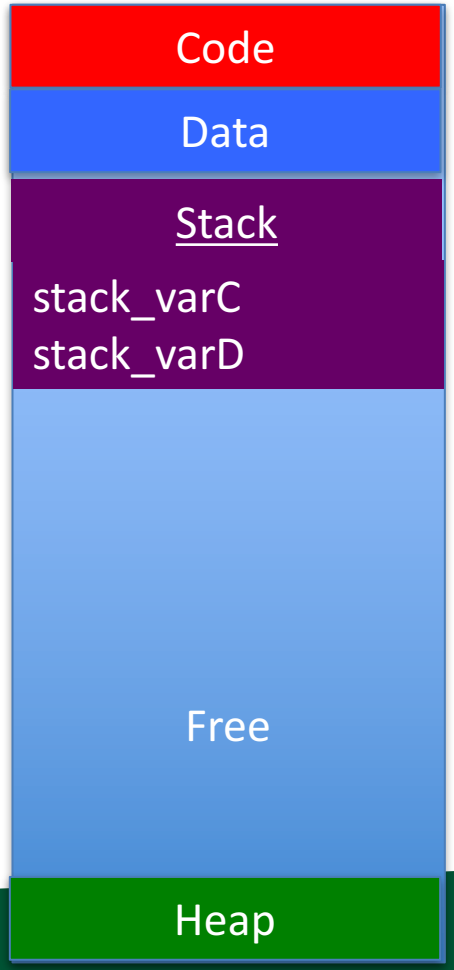
How stack memory is allocated into Stack Memory Segment

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

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



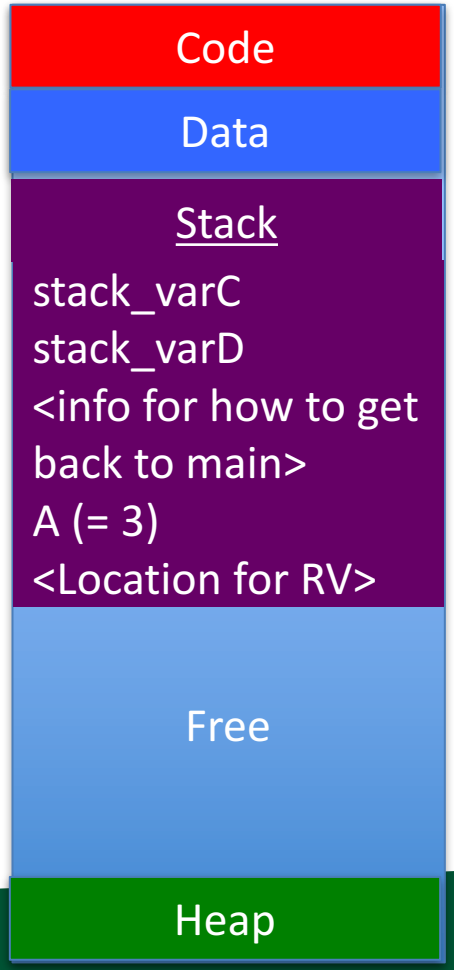
How stack memory is allocated into Stack Memory Segment

```
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);
}
```



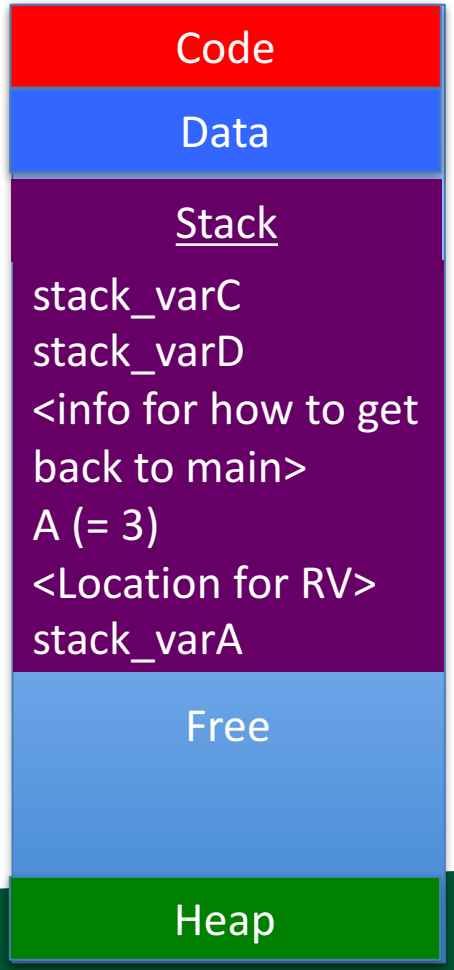
How stack memory is allocated into Stack Memory Segment

```
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

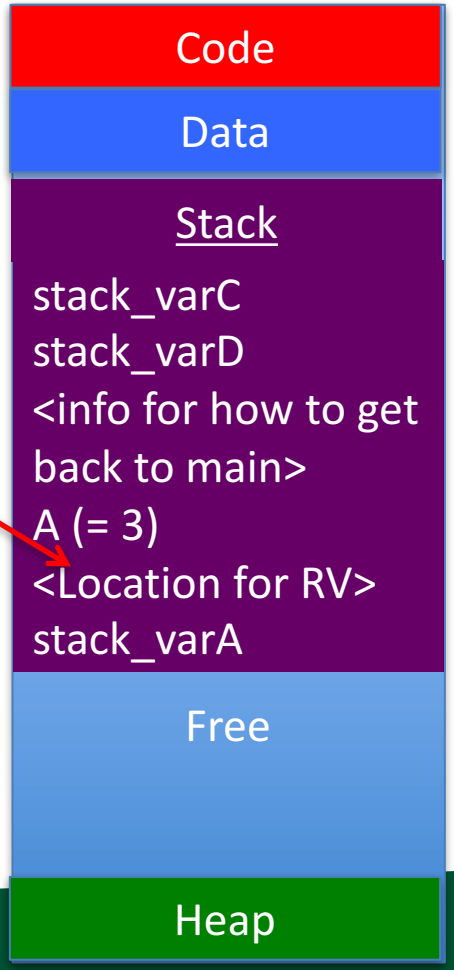
```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

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



How stack memory is allocated into Stack Memory Segment

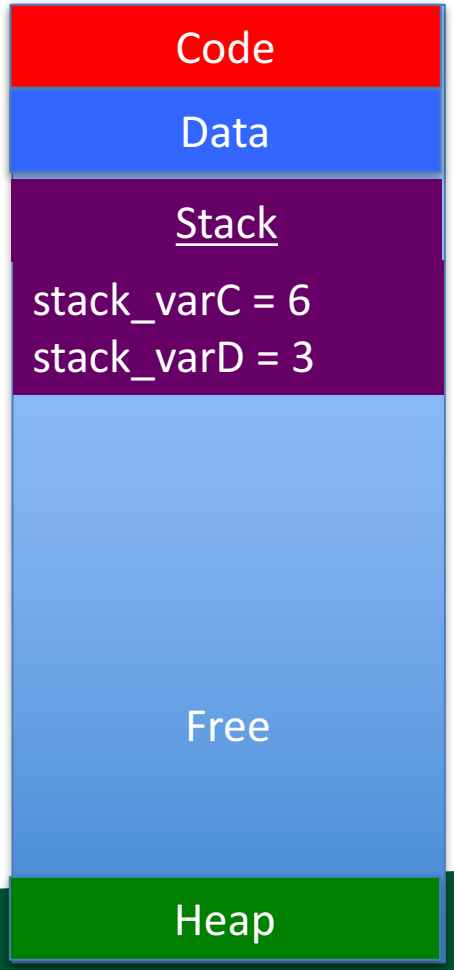
```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

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);
}
```



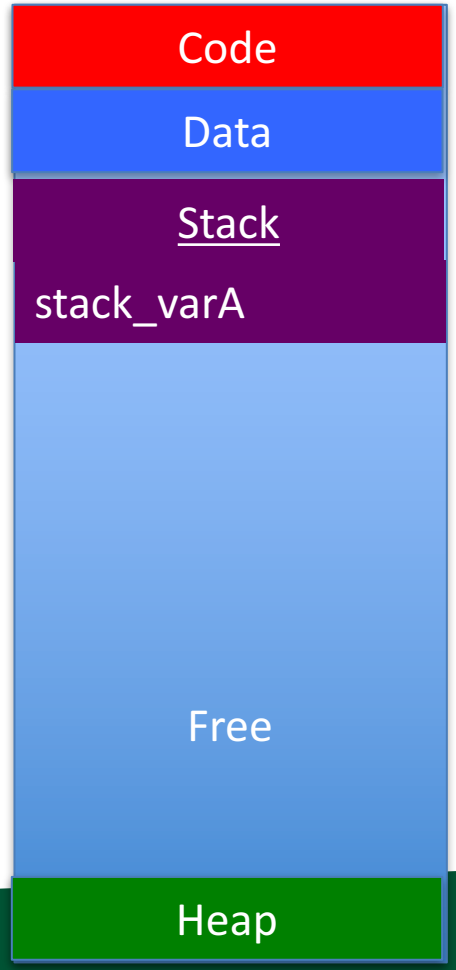
This code is very problematic … why?

```
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)

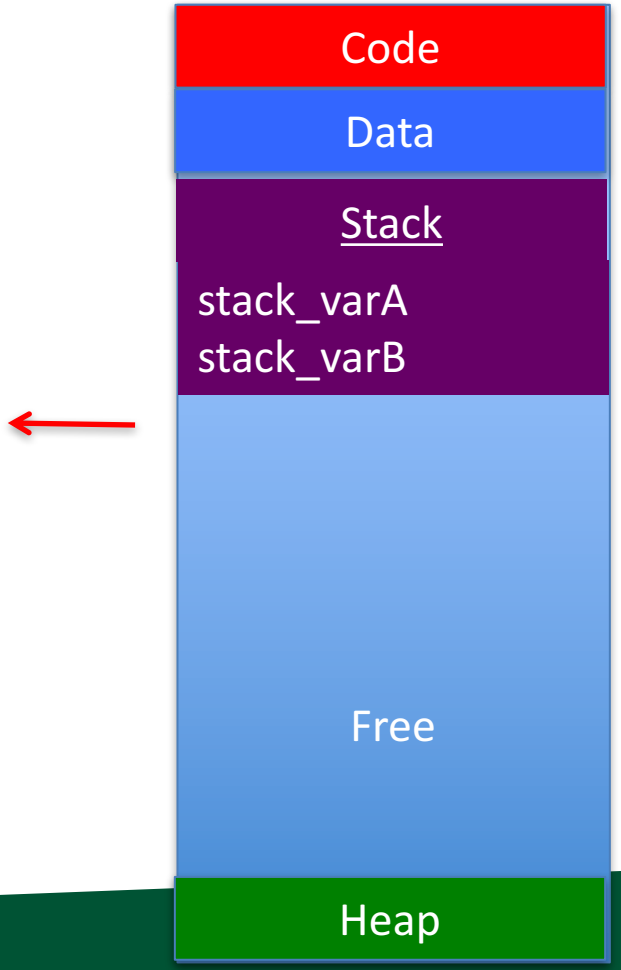
Nested Scope

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



Nested Scope

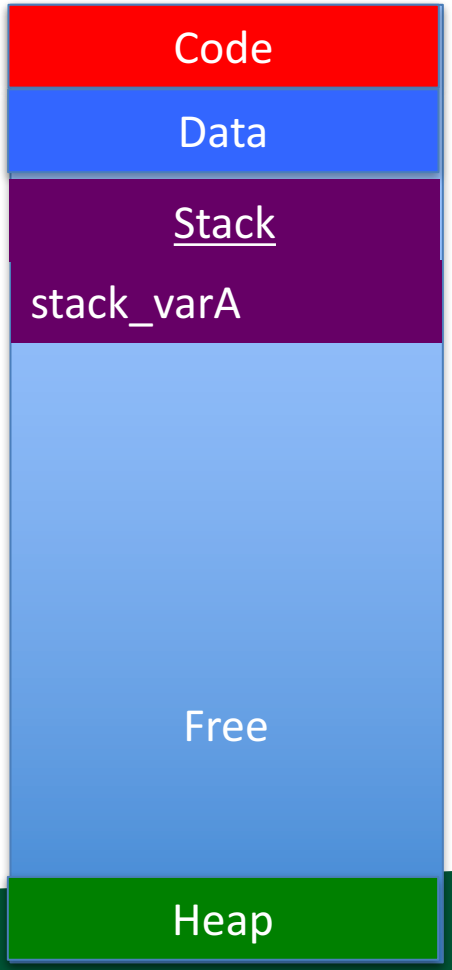
```
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```



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/Deal location	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/Deal location	Automatic	Explicit
Access	Fast	Slower
Variable scope	Limited	Unlimited

Variable scope: stack

```
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".

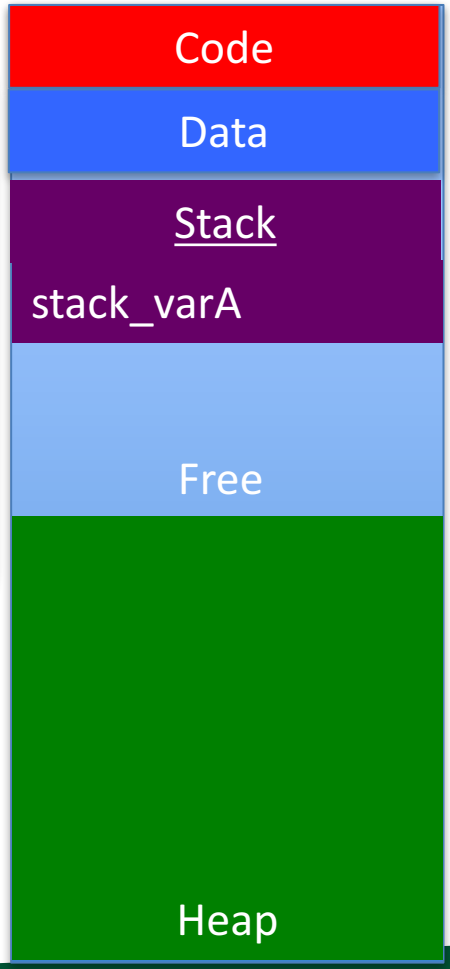
Memory leaks

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

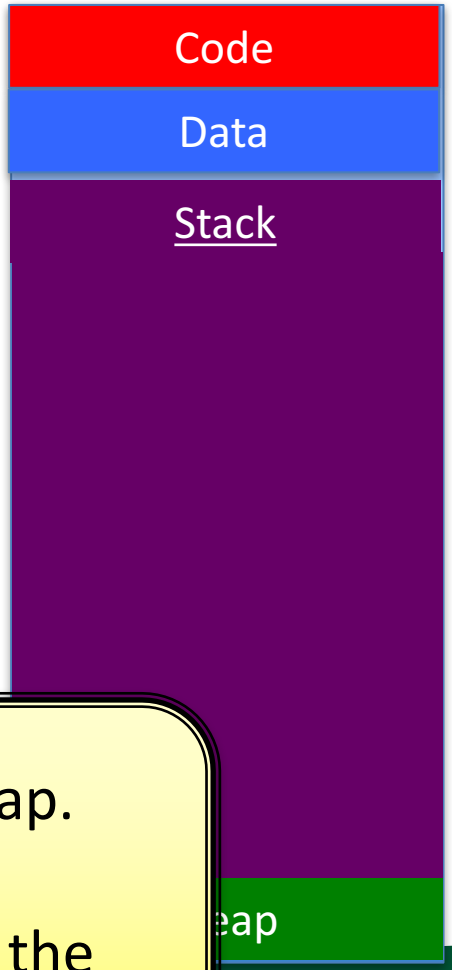
The heap's memory manager will not be able to re-claim them … we have effectively limited the memory available to the program.

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



Running out of memory (stack)

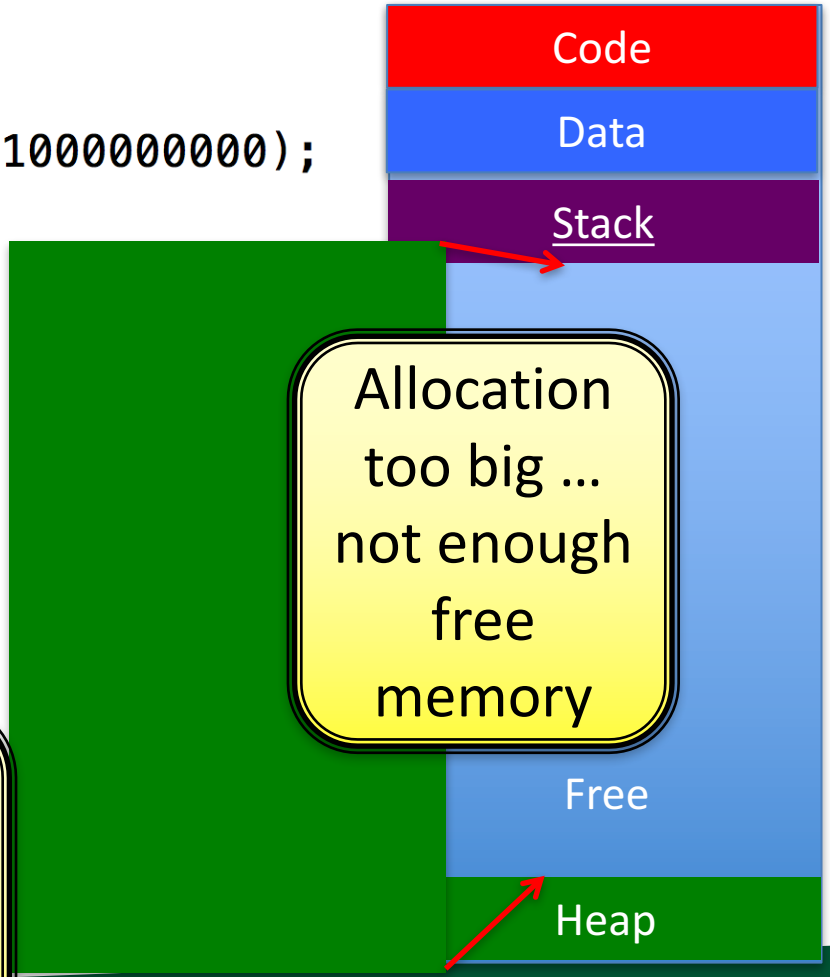
```
int endless_fun()
{
    endless_fun();
}

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



stack overflow: when the stack runs into the heap.
There is no protection for stack overflows.
(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();
}
```



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/Deal location	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]);
}
```

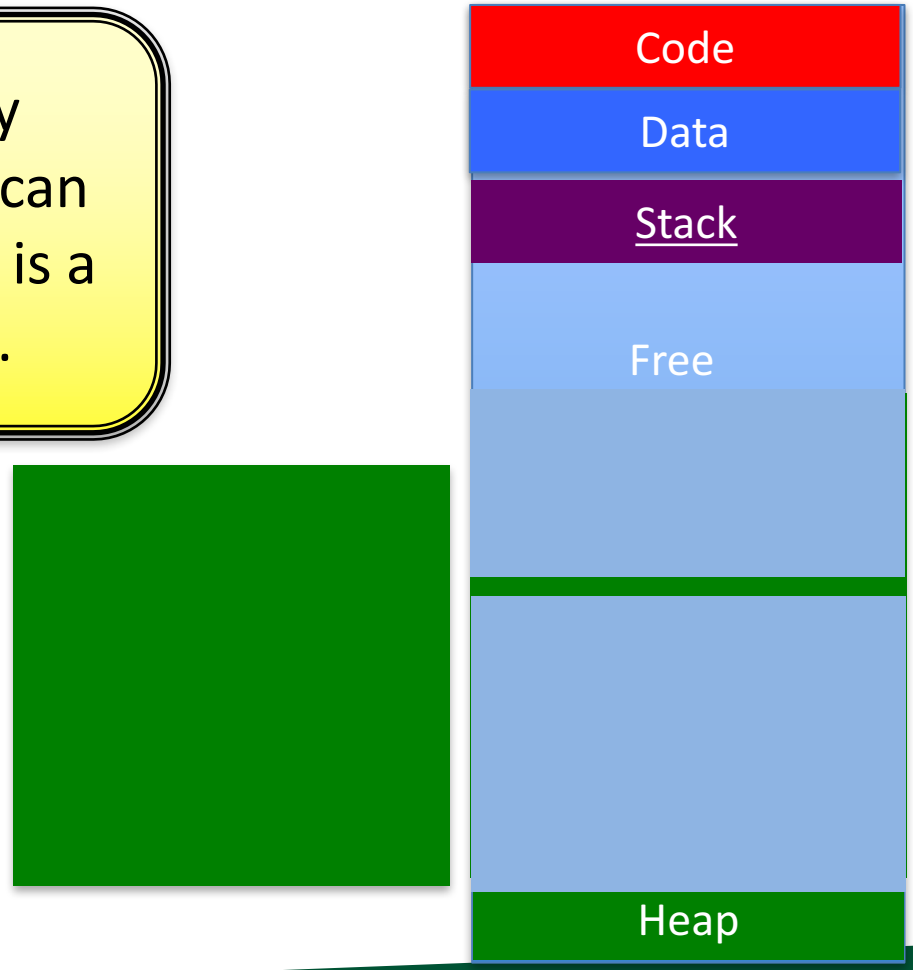
Negative aspects of fragmentation?

(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/Deal location	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];
}
```

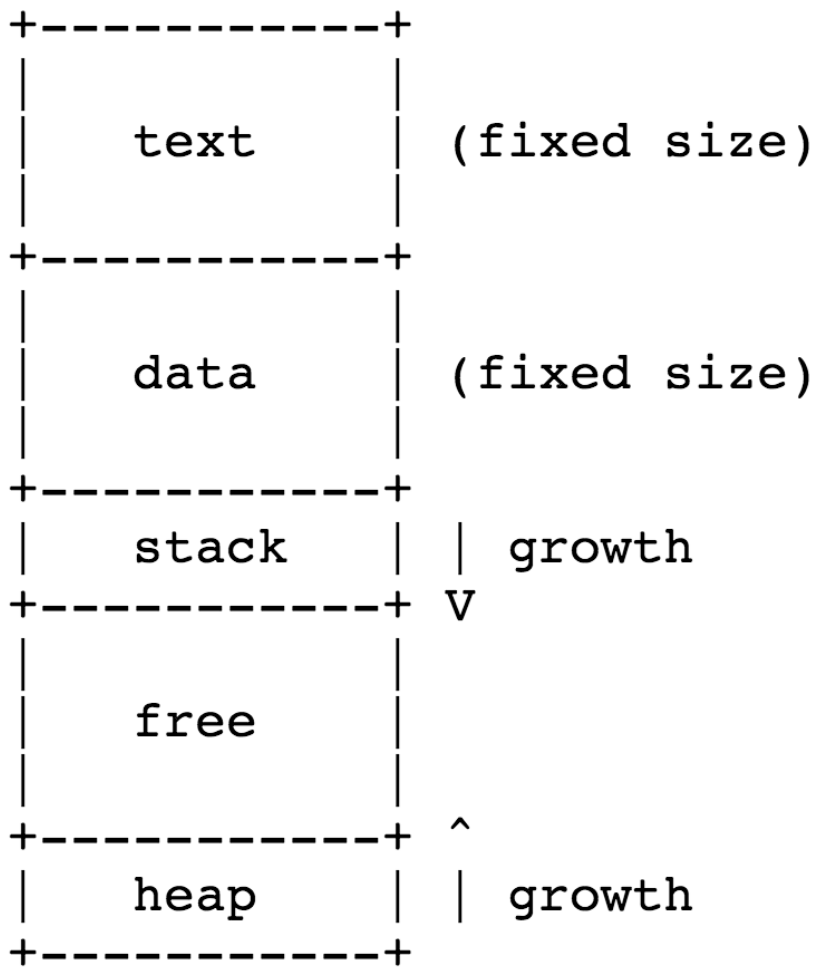
- Array bounds write

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

Outline

- Permissions
- Project 1B Overview
- More on memory / arrays / pointers

Memory Segments



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

C: must manage your own memory

- This is a big change from other programs
- You keep track of memory
  - Allocation
  - How much there is / indexing memory
  - Deallocation

malloc

- malloc: command for allocating memory

```
MALLOC(3)                BSD Library Functions Manual                MALLOC(3)

NAME
     calloc, free, malloc, realloc, reallocf, valloc -- memory allocation

SYNOPSIS
     #include <stdlib.h>

     void *
     calloc(size_t count, size_t size);

     void
     free(void *ptr);

     void *
     malloc(size_t size);

     void *
     realloc(void *ptr, size_t size);

     void *
     reallocf(void *ptr, size_t size);

     void *
     valloc(size_t size);

DESCRIPTION
     The malloc(), calloc(), valloc(), realloc(), and reallocf() functions allocate memory.  The allocated memory is
     aligned such that it can be used for any data type, including AltiVec- and SSE-related types.  The free() function
     frees allocations that were created via the preceding allocation functions.

     The malloc() function allocates size bytes of memory and returns a pointer to the allocated memory.
```

Allocation / Deallocation Example

```
#include <stdlib.h>
int main()
{
    int stack_varA;
    int stack_varB[2];
    int *heap_varA;
    int *heap_varB;
    heap_varA = malloc(sizeof(int));
    heap_varB = malloc(sizeof(int)*2);
    free(heap_varA);
    free(heap_varB);
}
```

Automatic allocation on the stack. (Deallocation occurs when out of scope.)

Explicit allocation from the heap. (Deallocation occurs with “free” call.)

sizeof

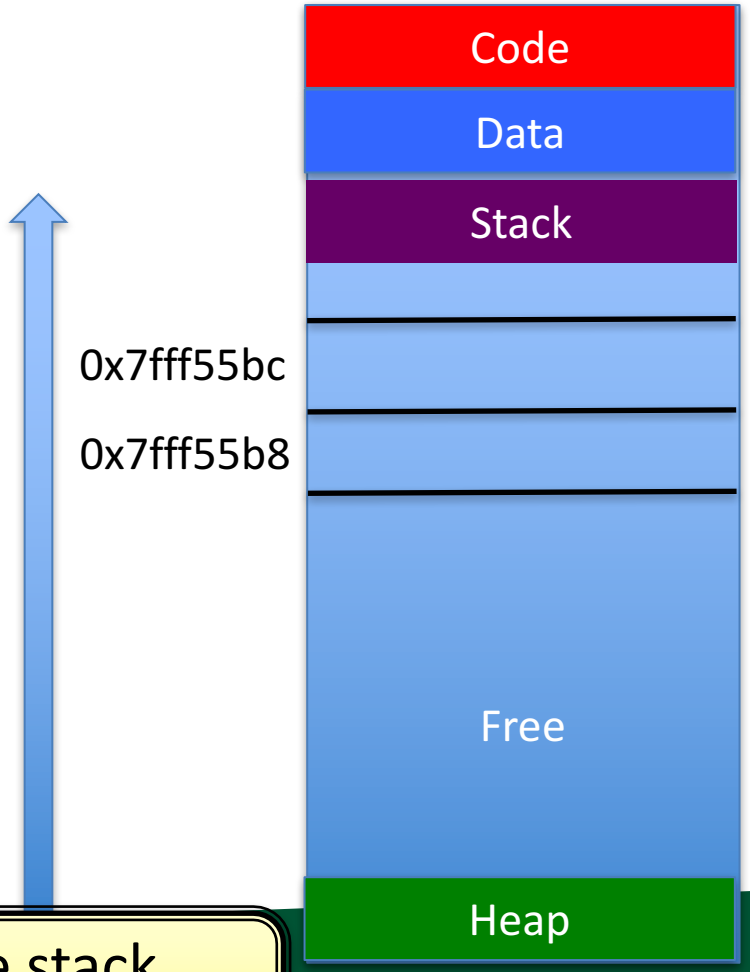
- sizeof: gets size of type
- Usually:
  - sizeof(int) == 4
  - sizeof(float) == 4
  - sizeof(double) == 8
  - sizeof(unsigned char) == 1
  - sizeof(char) == 1
  - sizeof(int *) == sizeof(double *) == sizeof(char *) == 8
- → array of 10 ints → malloc(10*sizeof(int))

Hexadecimal

- Binary: 2 values
- Decimal: 10 values
- Hexadecimal: 16 values
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- 0x: prefix for hexadecimal
- 0x10 = 16
- 0x101 = 257

Memory Addresses

- Every location in memory has an address associated with it
- Locations in memory are represented in hexadecimal



Memory addresses descend in the stack, ascend in the heap.

Pointers

- Pointers store locations in memory

Pointers

- Pointers store locations in memory

- “&”: unary operator that gives the address of a variable.

```
int x;
int *yp = &x;
```

Pointers

- Pointers store locations in memory

```
C02LN00GFD58:330 hank$ cat pointer.c
#include <stdio.h>

int main()
{
    int x, y;
    printf("The location of x is %p and the location of y is %p\n", &x, &y);
}
C02LN00GFD58:330 hank$ gcc pointer.c
C02LN00GFD58:330 hank$ ./a.out
The location of x is 0x7fff56d26bcc and the location of y is 0x7fff56d26bc8
```

printf prints pointers with “%p”

NULL pointer

- NULL: defined by compiler to be a location that is not valid.
  - Typically 0x00000000
- You can use NULL to initialize pointers, and also to check to see whether a pointer is set already.

IBM team I worked on used 0xDEADBEEF, not NULL

```
C02LN00GFD58:330 hank$ cat null.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *ptr = NULL;
    while (1)
    {
        char c;
        c = getchar();
        if (c == 'A')
        {
            if (ptr == NULL)
            {
                printf("Allocating!\n");
                ptr = malloc(100*sizeof(int));
            }
            else
                printf("Already allocated\n");
        }
    }
}
C02LN00GFD58:330 hank$ gcc null.c
C02LN00GFD58:330 hank$ ./a.out
A
Allocating!
A
dy allocated

dy allocated
```

'*' operator

- Let "ptr" be a pointer
- Then "*ptr" returns value in the address that ptr points to.
- * = "dereference operator"

```
C02LN00GFD58:330 hank$ cat ptr.c
#include <stdio.h>

int main()
{
    int x = 3;
    int *y = &x;
    int z = *y;
    printf("x = %d, z = %d\n", x, z);
}
C02LN00GFD58:330 hank$ gcc ptr.c
C02LN00GFD58:330 hank$ ./a.out
x = 3, z = 3
```

Behavior of dereference

- When you dereference, you get the value at that moment.
  - Whatever happens afterwards won't have effect.

```
C02LN00GFD58:330 hank$ cat ptr2.c
#include <stdio.h>

int main()
{
    int x = 3;
    int *y = &x;
    int z = *y;
    x = 4;
    printf("x = %d, y = %d, z = %d\n", x, *y, z);
}
C02LN00GFD58:330 hank$ gcc ptr2.c
C02LN00GFD58:330 hank$ ./a.out
x = 4, y = 4, z = 3
```

Pointer Arithmetic

- You can combine pointers and integers to get new pointer locations

```
C02LN00GFD58:330 hank$ cat ptr_arith.c
#include <stdio.h>

int main()
{
   int x = 3;
   int *y = &x;
   int *z = y+1;
   char a = 'A';
   char *b = &a;
   char *c = b+1;
   printf("x = %d, y = %p, z = %p\n", x, y, z);
   printf("a = %c, b = %p, c = %p\n", a, b, c);
}
C02LN00GFD58:330 hank$ gcc ptr_arith.c
C02LN00GFD58:330 hank$ ./a.out
x = 3, y = 0x7fff5d397bcc, z = 0x7fff5d397bd0
a = A, b = 0x7fff5d397bb7, c = 0x7fff5d397bb8
```

ptr + 1 → ptr + sizeof(type) bytes

Arrays

- Arrays: container that has multiple elements of identical type, all stored in contiguous memory

int A[10];

→ 10 integers, stored in 40 consecutive bytes (assuming sizeof(int) == 4)

Arrays are just pointers. You can use arrays and pointers interchangeably.

[ ] operator

- [ ] is a way of dereferencing memory
  - Recall that '*' is the dereference operator

- A[0] <= => *A
- A[5] <= => *(A+5);

More array relationships

```
int A[10];
int *B;
```

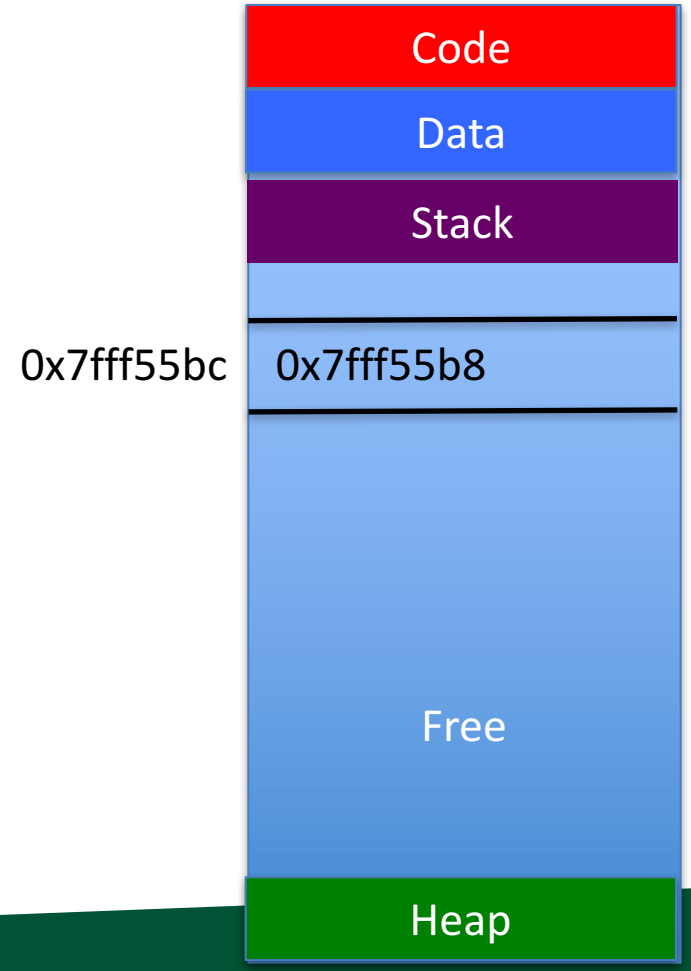
B=(A+5) → A[5] = B[0]

B=&(A[0]) → B = A

B=&(A[5]) → B = A+5

Pointers to pointers

- Remember: pointer points to a location in memory
  - We've been considering cases where locations in memory are arrays of integers
  - But locations in memory could be pointer themselves



Simple pointers to pointers example

```
C02LN00GFD58:330 hank$ cat ptrptr.c
#include <stdlib.h>
int main()
{
    int **X = malloc(sizeof(int *)*4);
    X[0] = malloc(sizeof(int)*6);
    X[1] = malloc(sizeof(int)*4);
    X[2] = malloc(sizeof(int)*8);
    X[3] = malloc(sizeof(int)*10);
}
C02LN00GFD58:330 hank$ gcc ptrptr.c
C02LN00GFD58:330 hank$ ./a.out
```

What's the difference between these two programs?

```
C02LN00GFD58:330 hank$ cat ptrptr.c
#include <stdlib.h>
int main()
{
    int **X = malloc(sizeof(int *)*4);
    X[0] = malloc(sizeof(int)*6);
    X[1] = malloc(sizeof(int)*4);
    X[2] = malloc(sizeof(int)*8);
    X[3] = malloc(sizeof(int)*10);
}
C02LN00GFD58:330 hank$ gcc ptrptr.c
C02LN00GFD58:330 hank$ ./a.out
```

```
C02LN00GFD58:330 hank$ cat ptrptr2.c
#include <stdlib.h>
int main()
{
    int *X[4];
    X[0] = malloc(sizeof(int)*6);
    X[1] = malloc(sizeof(int)*4);
    X[2] = malloc(sizeof(int)*8);
    X[3] = malloc(sizeof(int)*10);
}
C02LN00GFD58:330 hank$ gcc ptrptr2.c
C02LN00GFD58:330 hank$ ./a.out
```

Answer: X is on the heap on the left, and on the stack on the right. But they are both pointers-to-pointers.

What's the difference between these two programs?

```
C02LN00GFD58:330 hank$ cat ptrptr.c
#include <stdlib.h>
int main()
{
    int **X = malloc(sizeof(int *)*4);
    X[0] = malloc(sizeof(int)*6);
    X[1] = malloc(sizeof(int)*4);
    X[2] = malloc(sizeof(int)*8);
    X[3] = malloc(sizeof(int)*10);
}
C02LN00GFD58:330 hank$ gcc ptrptr.c
C02LN00GFD58:330 hank$ ./a.out
```

```
C02LN00GFD58:330 hank$ cat ptrptr3.c
#include <stdlib.h>
int main()
{
    int **X = malloc(sizeof(int *)*4);
    int num = 6+4+8+10;
    int *allMem = malloc(sizeof(int)*num);
    X[0] = allMem;
    X[1] = X[0]+6;
    X[2] = X[1]+4;
    X[3] = X[2]+8;
}
C02LN00GFD58:330 hank$ gcc ptrptr3.c
C02LN00GFD58:330 hank$ ./a.out
```

Answer: program on left makes one allocation for each pointer, program on right makes one allocation for whole program & each pointer points at locations within that allocation.

Call by value / call by reference

- Refers to how parameters are passed to a function.
  - Call by value: send the value of the variable as a function parameter
    - Side effects in that function don't affect the variable in the calling function
  - Call by reference: send a reference (pointer) as a function parameter
    - Side effects in that function affect the variable in the calling function

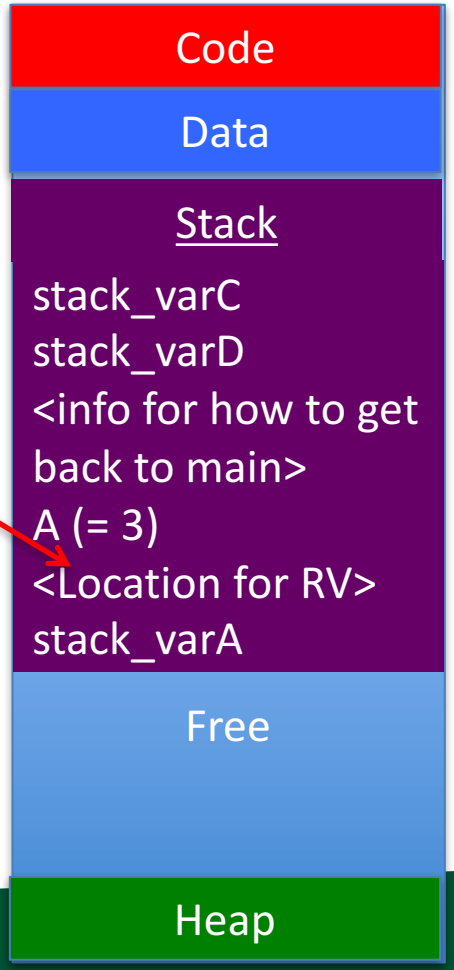
Call by Value

```
C02LN00GFD58:330 hank$ cat cbv.c
#include <stdio.h>

void foo(int x)
{
    x = x+1;
}

int main()
{
    int x = 2;
    foo(x);
    printf("X is %d\n", x);
}
C02LN00GFD58:330 hank$ gcc cbv.c
C02LN00GFD58:330 hank$ ./a.out
X is 2
```

Call by value

```
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



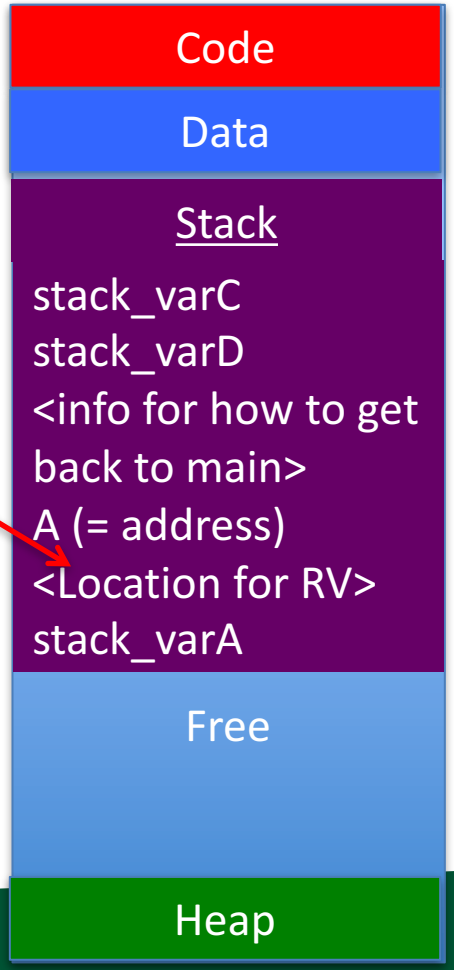
Call by reference

```
C02LN00GFD58:330 hank$ cat cbr.c
#include <stdio.h>

void foo(int *x)
{
    *x = *x+1;
}

int main()
{
    int x = 2;
    foo(&x);
    printf("X is %d\n", x);
}
C02LN00GFD58:330 hank$ gcc cbr.c
C02LN00GFD58:330 hank$ ./a.out
X is 3
```

Call by reference

```
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



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?