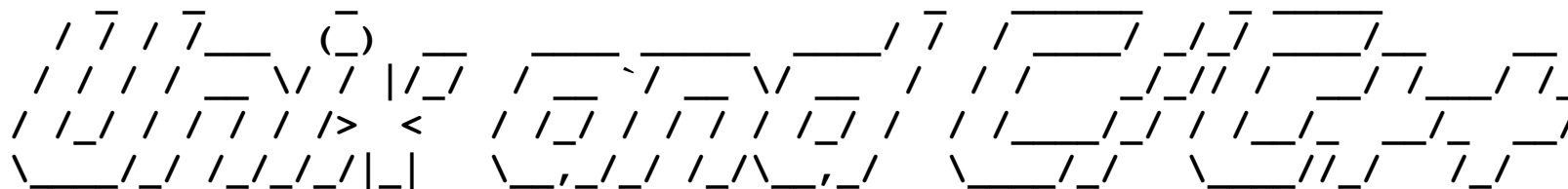




CIS 607:



Lecture 1.1: Course Overview & Introduction to Unix



Outline

- Class Overview
- Getting Started With Unix
 - Unix History
 - Shell Prompts
 - Files
 - File Editors
- Project 1



COVID / Illness

- If you have to quarantine or feel sick, then:
 - (1) email me ahead of time
 - (2) call in to the Zoom
 - Zoom listed on Canvas site
 - (3) you will be credited with attending
- Please, please stay home if there is a possibility of spreading COVID



Why Does This Seminar Exist?

- Some schools make their undergraduate students do a lot of Unix/C/C++ preparation.
- Some do not.
- At UO, we assume our grad students know Unix/C/C++.
- If you didn't get a lot of preparation, this is a chance to do it now.
- (Also a good refresher for those interviewing for jobs)
- Note: this course draws a lot of material from previous courses. Apologies in advance.
- Note #2: material is pretty basic this week. Will get more advanced.



Course Derived from CIS 330

- 330 Goals: excellence in C, C++, and Unix
- Why 607?
 - Many of our grad-level classes require strong knowledge in C, C++, and Unix
 - Critical for success after graduation
- Programming Languages Beacon:
<http://www.lextrait.com/vincent/implementations.html>



Grading For This Course (1/2)

- Will assign ~18 projects
- Only 2 will be graded: 3H and 4B
- → these two depend on 3A-3G
- The rest of the projects are to prepare you to do 3H and 4B



Grading For This Course (2/2)

Students can earn up to 100 points:

- Weekly attendance in lecture: 45 points (5 points per lecture, maximum of 45, even if attending all 10 weeks)
 - Students effectively can miss one lecture for free. This is intended for attending conferences, illness, family emergencies, etc. If students need to miss more than one lecture for such reasons, they should contact the instructor.
- Completing final projects (up to 55 points)
 - less than 30% of 3H tests passing: 0 points
 - between 30% and 69% of 3H tests passing: 20 points
 - between 70% and 99% of 3H tests passing: 35 points
 - 100% of 3H tests passing: 45 points
 - Project 4B passes memory error free: 5 points
 - Project 4B passes memory leak free: 5 points

Grading will not be curved:

- 95, 100 points: A
- 90 points: A-
- 85 points: B
- 80 points: B-
- 75 points: C
- 70 points: C-
- 65 points: D
- 60 points: D-
- less than 60 points: F



Norms for this class

- Please ask questions
- Please ask me to slow down
- Please give feedback
- Quiet classroom greatly valued
- Please do not arrive late



Course Materials

- PowerPoint lectures will be posted online.
- I will “live code” frequently.
- Textbook:
 - Past terms: none
 - This term: incorporating “C and Data Structures” by Sventek
 - On Canvas (legal statement next slide)



C and Data Structures - a well-structured approach

Joseph S. Sventek

This book contains copyrighted material. You may use it for this class under the following constraints:

"Permission is granted for one time classroom use for registered learners only. The duration of use is only for the duration of the course. The material may not be published and distributed outside of the course."

Thus, you may make a copy for your use on your own machine. You may NOT share this book with anyone outside of the class, nor may you post it ANYWHERE on the web, Facebook, or any other social media platform.

Failure to abide by these rules will lead to significant legal difficulties for the University, the Department, your instructor, and yourself.



Academic Misconduct (1 of 2)

- The programming projects are individual efforts
 - You may discuss the projects with your classmates.
 - Do not let someone look at your code on your screen. (BUT: helper can look at helpee's code)
 - Absolutely, positively do not email code.
 - Do not search the internet for previous implementations (includes github)



Academic Misconduct (2 of 2)

- If I detect collusion, all individuals involved will receive an F in the course immediately
 - I choose to not enumerate cases that involve collusion. Whiteboard conversations are fine. If appropriate, the helper can look at the helpee's code. If you feel you are in a gray area, then you should email me.
 - Please note that if you are the one providing too much help, then you will also get an F



IDEs

- IDEs are great
 - ... but in this class, we will learn how to get by without them
- Many, many Unix-based projects don't use IDEs
 - The skills you are using will be useful going forward in your careers



Accessing a Unix environment

- Rm 100, Deschutes
- Remote logins (ssh, scp)
- Windows options
 - Cygwin / MSYS
 - Virtual machines

Who has home access to a Unix environment?

Who has Windows?



Missed Class on Week 1 Is an Opportunity??

- This class: do 3A-3H, 4B
- But: need to get through a lot of lecture before you can begin
- Ends up being crowded at the end
- Different idea?
 - More lectures early in the term
 - Skip lectures later in the term



Outline

- Class Overview
- Getting Started With Unix
 - Unix History
 - Shells
 - Files
 - File Editors
- Project 1



Reading

- C and Data Structures:
 - Chapter 2.1, 2.2, 2.3, 2.4., 2.5, and 2.6.1.



Outline

- Class Overview
- Getting Started With Unix
 - Unix History
 - Shells
 - Files
 - File Editors
- Project 1



What is Unix?

- Operating system
 - Multi-tasking
 - Multi-user
- Started at AT&T Bell Labs in late '60s, early '70s
- First release in 1973



What is Unix?

- 80s & 90s: many competing versions, all conforming to same standard
 - AIX (IBM), Solaris (Sun), HP-UX (Hewlett-Packard)
- 1990s: Linux takes off
 - Open source
- 2000s: commercial Unixes abandoned, companies use Linux, back Linux
 - Several variants of Linux
- OS X: used on Macs since 2002
 - Meets Unix standard



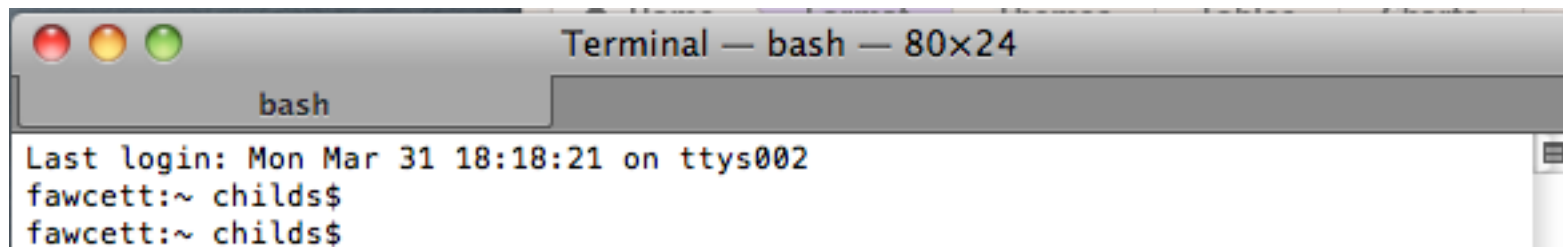
Outline

- Class Overview
- Getting Started With Unix
 - Unix History
 - Shells
 - Files
 - File Editors
- Project 1



Shells

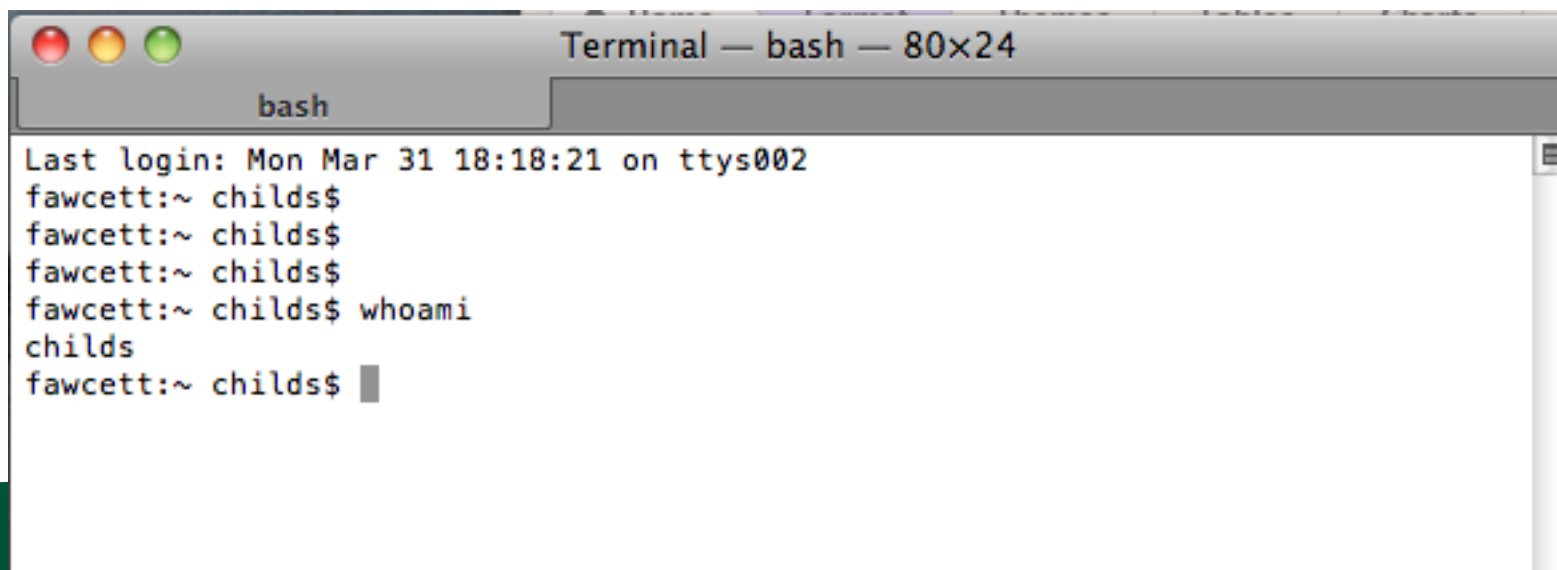
- Shells are accessed through a terminal program
 - Typically exposed on all Linux
 - Mac: Applications->Utilities->Terminal
 - (I always post this on the dock immediately upon getting a new Mac)

A screenshot of a macOS Terminal window. The title bar reads "Terminal — bash — 80x24". The window contains the following text:

```
bash
Last login: Mon Mar 31 18:18:21 on ttys002
fawcett:~ childs$
fawcett:~ childs$
```

Shells

- Shells are interpreters
 - Like Python
- You type a command, it carries out the command

A screenshot of a terminal window titled "Terminal — bash — 80x24". The window has a tab labeled "bash". The terminal output shows a login message: "Last login: Mon Mar 31 18:18:21 on ttys002". Below this, the prompt "fawcett:~ childs\$" is shown three times, followed by the command "whoami" and its output "childs". The prompt "fawcett:~ childs\$" is shown again with a cursor at the end.

```
Terminal — bash — 80x24
bash
Last login: Mon Mar 31 18:18:21 on ttys002
fawcett:~ childs$
fawcett:~ childs$
fawcett:~ childs$
fawcett:~ childs$ whoami
childs
fawcett:~ childs$ █
```



Shells

- There are many types of shells
- Two most popular:
 - sh (= bash & ksh)
 - csh (= tcsh)
- They differ in syntax, particularly for
 - Environment variables
 - Iteration / loops / conditionals

The examples in this course will use syntax for sh



Environment Variables

- Environment variables: variables stored by shell interpreter
- Some environment variables create side effects in the shell
- Other environment variables can be just for your own private purposes



Environment Variables

A terminal window titled "Terminal — bash — 86x28" with a tab labeled "bash". The terminal output shows the last login time and the execution of the 'export' command.

```
bash
Last login: Mon Mar 31 18:44:25 on ttys003
fawcett:~ childs$ export CIS330=fun
```

New commands: export, echo, env



Shells

- There is lots more to shells ... we will learn about them as we go through the quarter



Outline

- Class Overview
- Getting Started With Unix
 - Unix History
 - Shells
 - Files
 - File Editors
- Project 1



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
```



Anatomy of shell formatting

Machine name Current working directory Username

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

The diagram illustrates the components of a shell prompt. Three blue arrows point from labels above to parts of the prompt 'fawcett:~ childs\$'. The arrow from 'Machine name' points to 'fawcett:'. The arrow from 'Current working directory' points to '~'. The arrow from 'Username' points to 'childs\$'. Below the prompt, the output of the command 'echo \$HOME' is shown as '/Users/childs', with a horizontal line underneath.

- “~” (tilde) is shorthand for your home directory
 - You can use it when invoking commands

The shell formatting varies over Unix implementation and can be customized with environment variables.
(PS1, PS2, etc)



File manipulation



```
Last login: Tue Apr  1 04:56:14 on ttys005
```

New commands: mkdir, cd, touch, ls, rmdir, rm
Also, "*" is a wildcard that matches any filename



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

Why could time stamps be useful?



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 (discussed more 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



Outline

- Class Overview
- Getting Started With Unix
 - Unix History
 - Shells
 - Files
 - File Editors
- Project 1



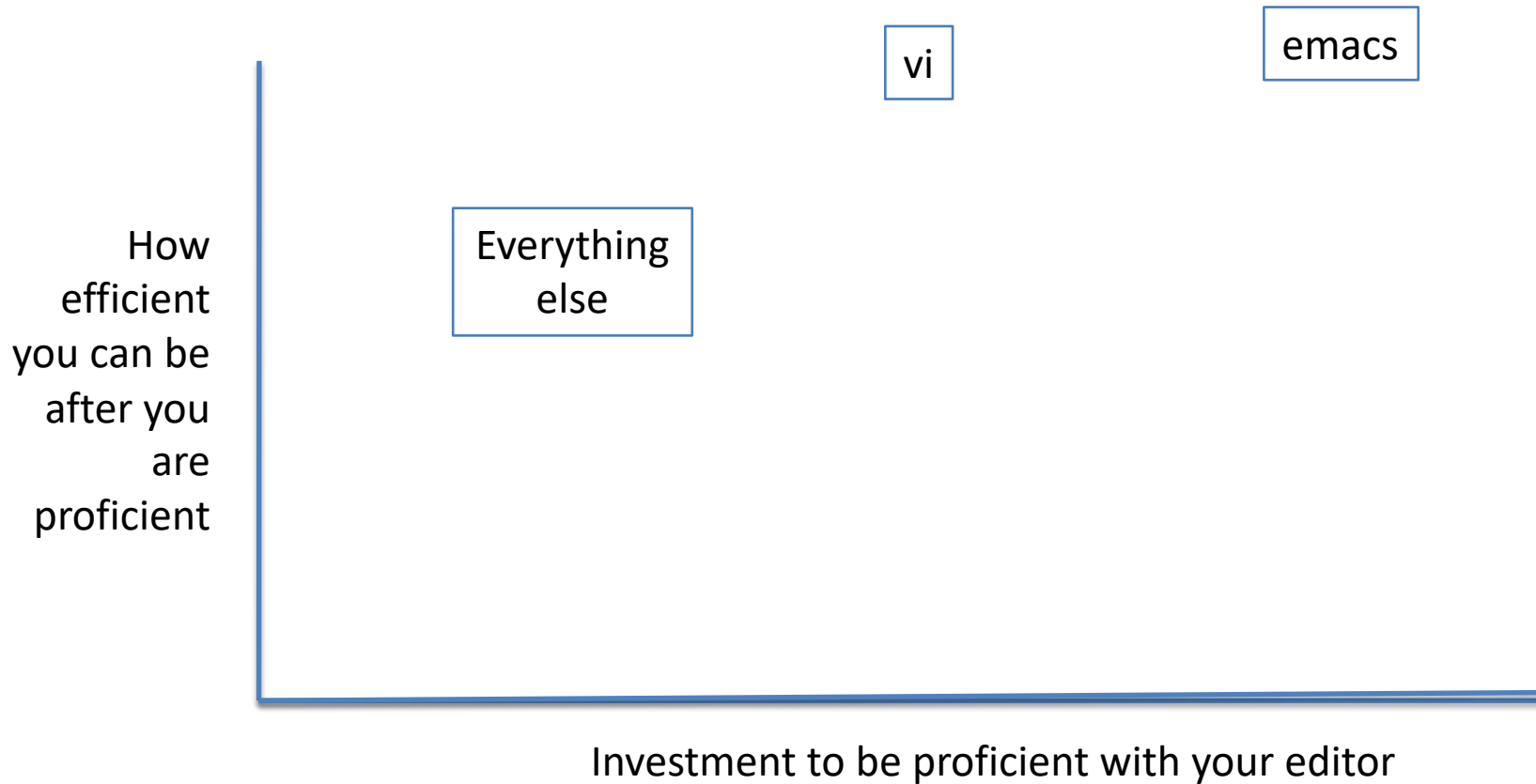
File Editors

- Existing file editors:
 - Vi
 - Emacs
 - Two or three hot new editors that everyone loves (and ultimately fade away and die)
- This has been the state of things for 25 years

I will use “vi” in this course.
You are welcome to use whatever editor you want.



My Mental Model for File Editors





Vi has two modes

- Command mode
 - When you type keystrokes, they are telling vi a command you want to perform, and the keystrokes don't appear in the file
- Edit mode
 - When you type keystrokes, they appear in the file.



Transitioning between modes

- Command mode to edit mode
 - i: enter into edit mode at the current cursor position
 - a: enter into edit mode at the cursor position immediately to the right of the current position
 - l: enter into edit mode at the beginning of the current line
 - A: enter into edit mode at the end of the current line

There are other ways to enter edit mode as well



Transitioning between modes

- Edit mode to command mode
 - Press Escape



Useful commands

- `yy`: yank the current line and put it in a buffer
 - `2yy`: yank the current line and the line below it
- `p`: paste the contents of the buffer
 - `2pp`: past the contents of the buffer two times
- `x`: delete the character at the current cursor
- `“:100”` go to line 100 in the file
- Arrows can be used to navigate the cursor position (while in command mode)
 - So do `h`, `j`, `k`, and `l`

We will discuss more tips for “vi” throughout the quarter. They will mostly be student-driven (Q&A time each class)



My first vi sequence

- At a shell, type: “vi cis330file”
- Press ‘i’ (to enter edit mode)
- Type “I am using vi and it is fun” (text appears on the screen)
- Press “Escape” (to enter command mode)
- Press “:wq” (command mode sequence for “write and quit”)

vi / vim graphical cheat sheet

Esc
normal mode

~ toggle case	! external filter	@ play macro	# prev ident	\$ eol	% goto match	^ "soft" bol	& repeat :s	* next ident	(begin sentence) end sentence	_ "soft" bol down	+ next line
\ goto mark	1	2	3	4	5	6	7	8	9	0 "hard" bol	- prev line	= auto ³ format
Q ex mode	W next WORD	E end WORD	R replace mode	T back 'till	Y yank line	U undo line	I insert at bol	O open above	P paste before	{ begin parag.	} end parag.	
q record macro	w next word	e end word	r replace char	t 'till	y yank ^{1,3}	u undo	i insert mode	o open below	p paste ¹ after	[misc] misc	
A append at eol	S subst line	D delete to eol	F "back" find ch	G eof/goto ln	H screen top	J join lines	K help	L screen bottom	. ex cmd line	" reg. spec ¹	bol/goto col	
a append	s subst char	d delete ^{1,3}	f find char	g extra ⁶ cmds	h ←	j ↓	k ↑	l →	. repeat ; t/T/ℓ/F	' goto mk. bol	\ not used!	
Z quit ⁴	X back-space	C change to eol	V visual lines	B prev WORD	N prev (find)	M screen mid'l	< un- ³ indent	> indent ³	? find (rev.)			
Z extra ⁵ cmds	X delete char	c change ^{1,3}	V visual mode	b prev word	n next (find)	m set mark	, reverse ; t/T/ℓ/F	. repeat cmd	/ find			

- motion** moves the cursor, or defines the range for an operator
 - command** direct action command, if **red**, it enters insert mode
 - operator** requires a motion afterwards, operates between cursor & destination
 - extra** special functions, requires extra input
 - q.** commands with a dot need a char argument afterwards
- bol = beginning of line, eol = end of line, mk = mark, yank = copy
- words: `quux(foo, bar, baz);`
 WORDS: `quux(foo, bar, baz);`

Main command line commands ('ex'):
 :w (save), :q (quit), :q! (quit w/o saving)
 :e f (open file f),
 :%s/x/y/g (replace 'x' by 'y' filewide),
 :h (help in vim), :new (new file in vim),

Other important commands:
 CTRL-R: redo (vim),
 CTRL-F/-B: page up/down,
 CTRL-E/-Y: scroll line up/down,
 CTRL-V: block-visual mode (vim only)

Visual mode:
 Move around and type operator to act on selected region (vim only)

- Notes:**
- (1) use "x before a yank/paste/del command to use that register ('clipboard') (x=a..z,*) (e.g.: "ay\$ to copy rest of line to reg 'a')
 - (2) type in a number before any action to repeat it that number of times (e.g.: 2p, d2w, 5i, d4j)
 - (3) duplicate operator to act on current line (dd = delete line, >> = indent line)
 - (4) ZZ to save & quit, ZQ to quit w/o saving
 - (5) zt: scroll cursor to top, zb: bottom, zz: center
 - (6) gg: top of file (vim only), gf: open file under cursor (vim only)

For a graphical vi/vim tutorial & more tips, go to www.viemu.com - home of ViEmu, vi/vim emulation for Microsoft Visual Studio



vimtutor

- Past students have liked vimtutor



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)

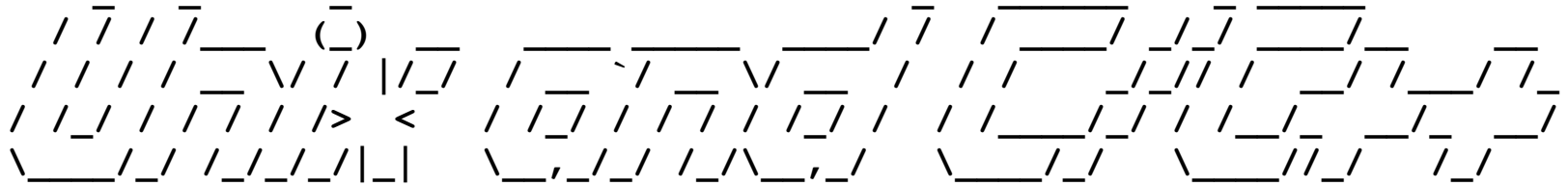
```
fawcett:~ childs$ vi hanks_essay
fawcett:~ childs$ wc -w hanks_essay
    252 hanks_essay
fawcett:~ childs$ wc hanks_essay
    63     252    1071 hanks_essay
fawcett:~ childs$ █
```

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



Don't forget

- This lecture is available online
 - <http://ix.cs.uoregon.edu/~hank/607>
- All project prompts are available online



Lecture 1.2: Memory in C



Plan for today

- 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

Unix command that prints contents of a file



```
C02LN00GFD58:CIS330 hank$ cat t.c
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    printf("hello world!\n");
```

```
}
```

Invoke gcc compiler

Name of file to compile

```
C02LN00GFD58:CIS330 hank$ gcc t.c
```

```
C02LN00GFD58:CIS330 hank$ ./a.out
```

```
hello world!
```

```
C02LN00GFD58:CIS330 hank$ █
```

gcc's default name for output programs

You should use this for Proj 2A



Plan for today

- 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.

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

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

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

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 dereferencing
 - `*` = “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

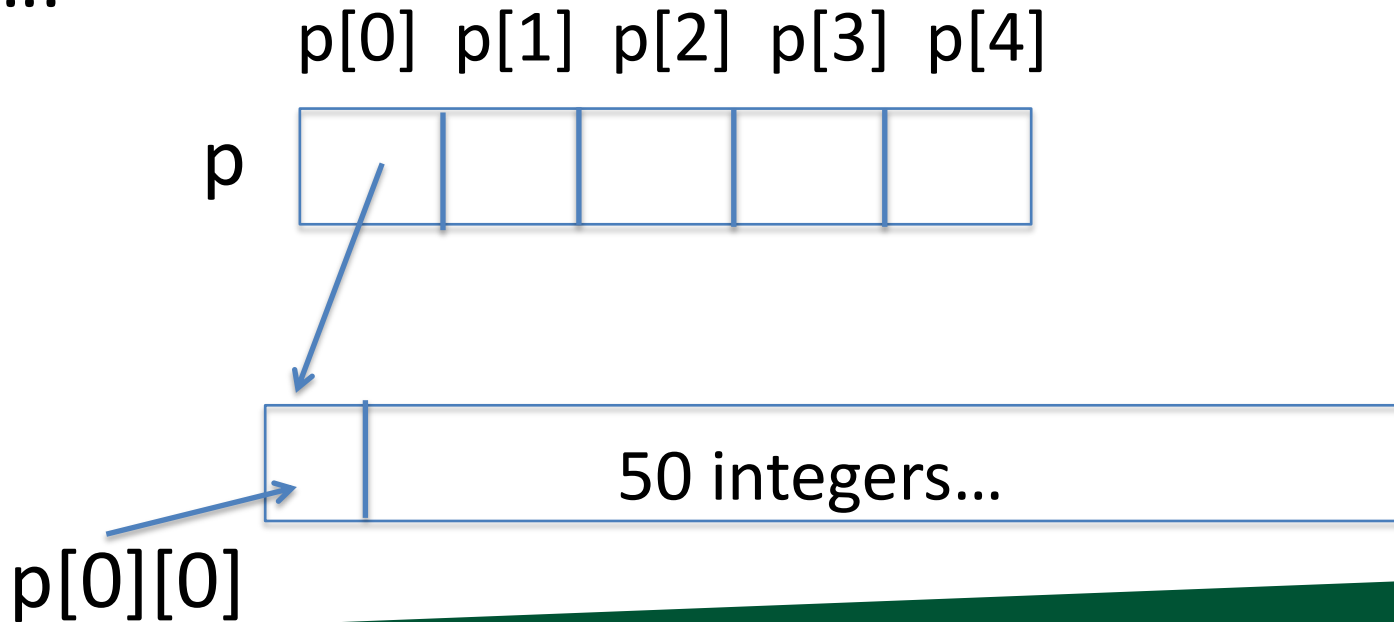
- `int *p = malloc(sizeof(int)*5);`
- C/C++ allows you to modify pointer with math operations
 - called pointer arithmetic
 - “does the right thing” with respect to type
 - `int *p = malloc(sizeof(int)*5);`
 - `p+1` is 4 bytes bigger than `p`!!
- Then:
 - “`p+3`” is the same as “`&(p[3])`” (ADDRESSES)
 - “`*(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 `0x00000000`
- 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**
- 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

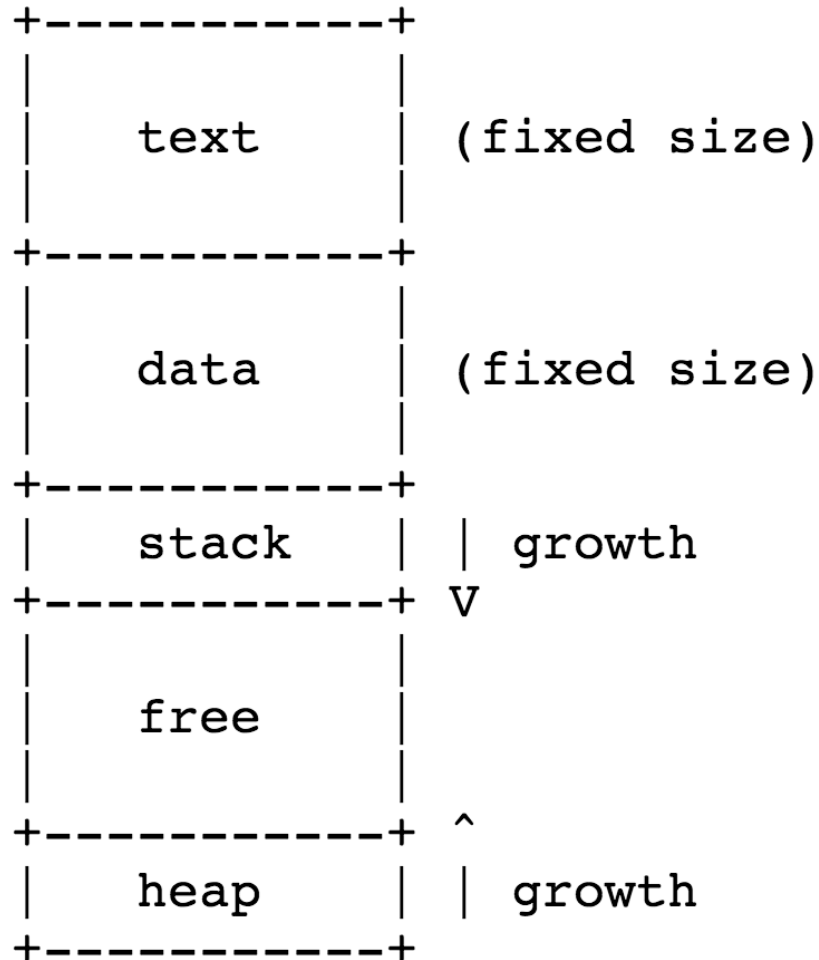


Heap

- 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





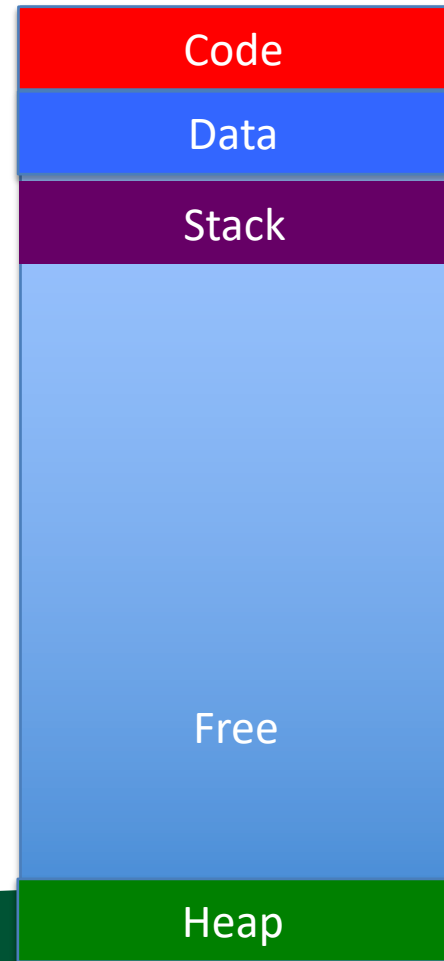
Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/Deallocation location	Automatic	Explicit



How stack memory is allocated into Stack Memory Segment

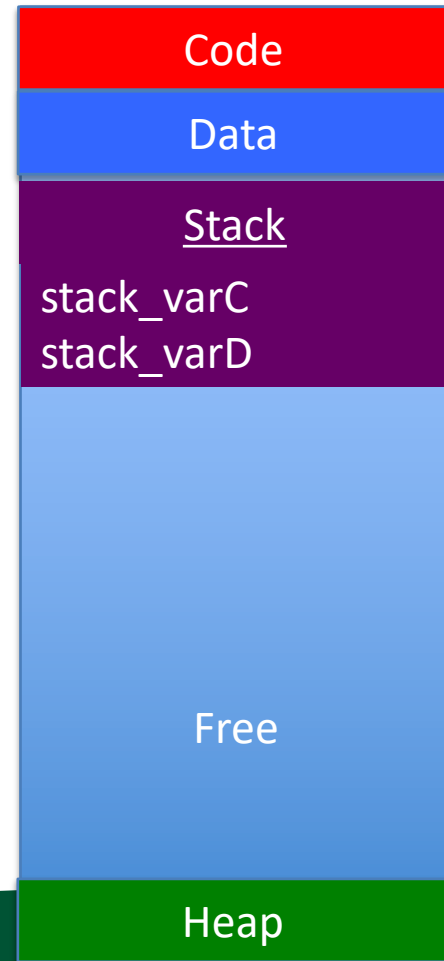
```
void foo()  
{  
    int stack_varA;  
    int stack_varB;  
}  
█  
int main()  
{  
    int stack_varC;  
    int stack_varD;  
    foo();  
}
```





How stack memory is allocated into Stack Memory Segment

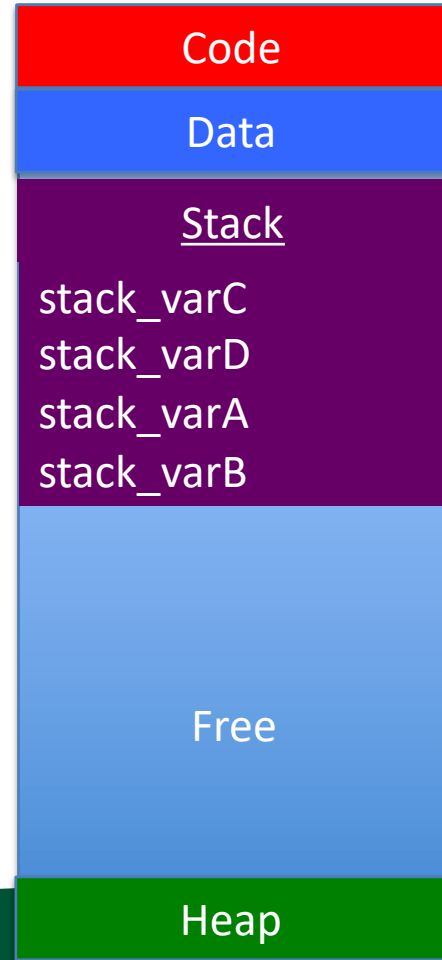
```
void foo()  
{  
    int stack_varA;  
    int stack_varB;  
}  
int main() ←  
{  
    int stack_varC;  
    int stack_varD;  
    foo();  
}
```





How stack memory is allocated into Stack Memory Segment

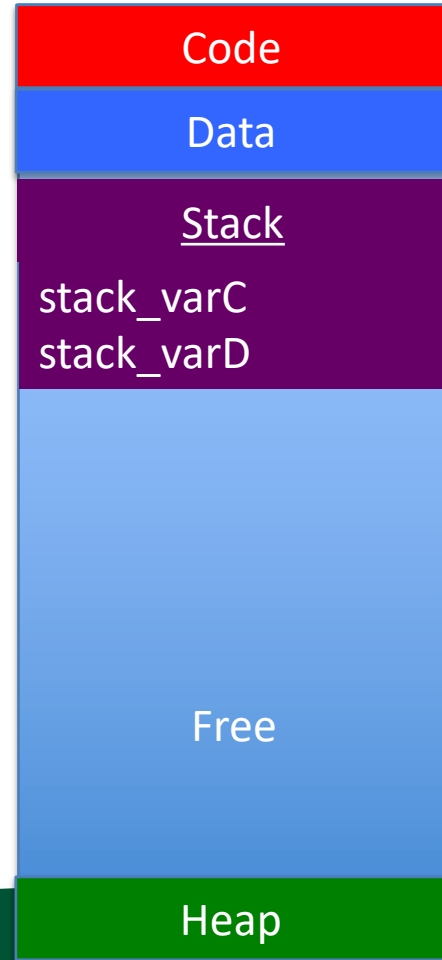
```
void foo() ←  
{  
    int stack_varA;  
    int stack_varB;  
}  
█  
int main()  
{  
    int stack_varC;  
    int stack_varD;  
    foo(); ←  
}
```





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();  
}
```

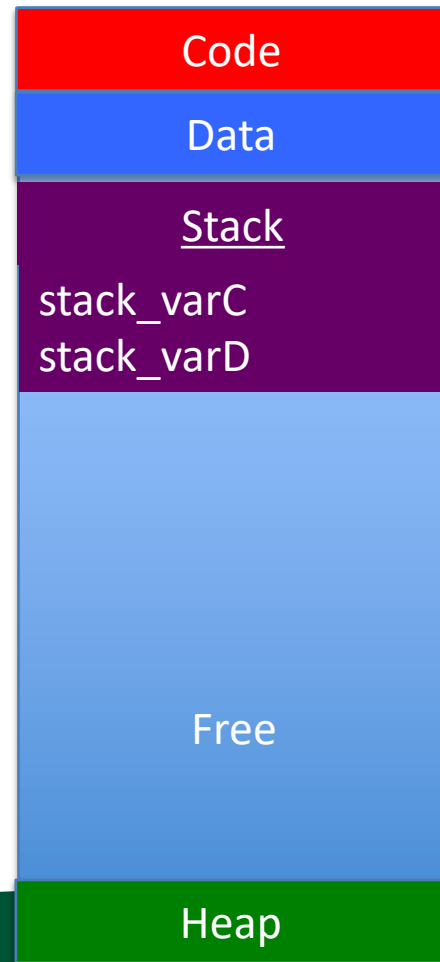




How stack memory is allocated into Stack Memory Segment

```
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);
}
```

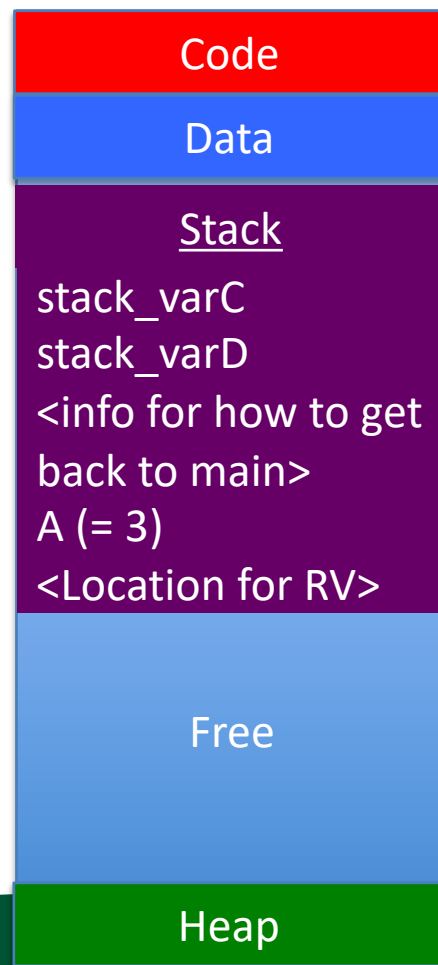




How stack memory is allocated into Stack Memory Segment

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

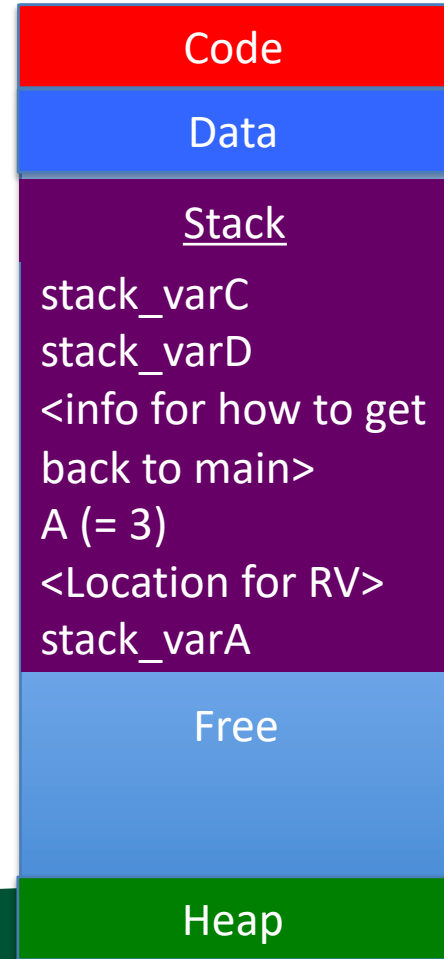
int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```





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);  
}
```



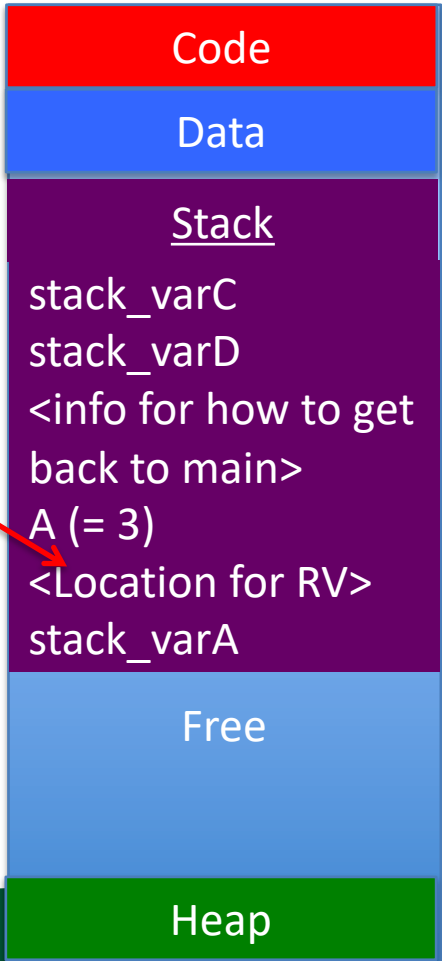


How stack memory is allocated into Stack Memory Segment

```
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

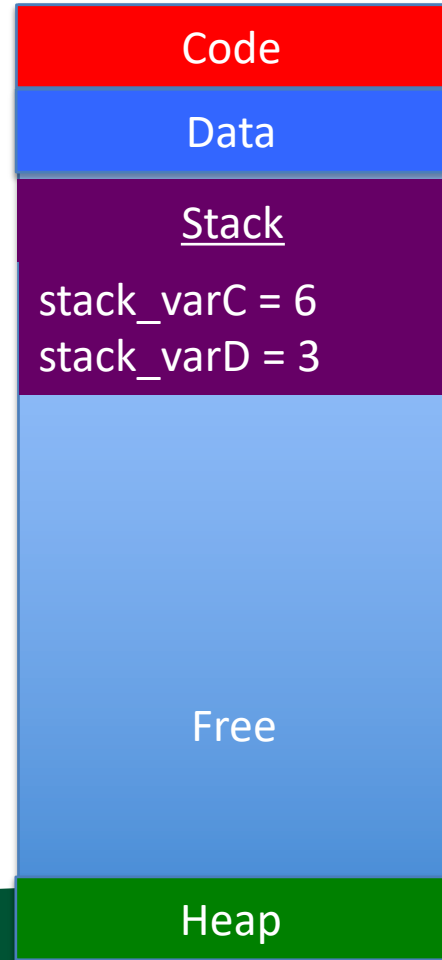




How stack memory is allocated into Stack Memory Segment

```
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?

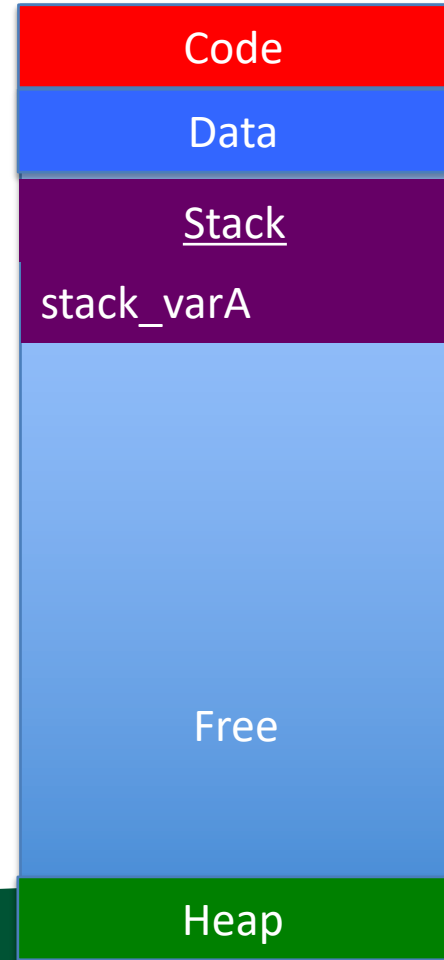
```
int *foo()
{
    int stack_varC[2] = { 0, 1 };
    return stack_varC;
}
int *bar()
{
    int stack_varD[2] = { 2, 3 };
    return stack_varD;
}
int main()
{
    int *stack_varA, *stack_varB;
    stack_varA = foo();
    stack_varB = bar();
    stack_varA[0] *= stack_varB[0];
}
```

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

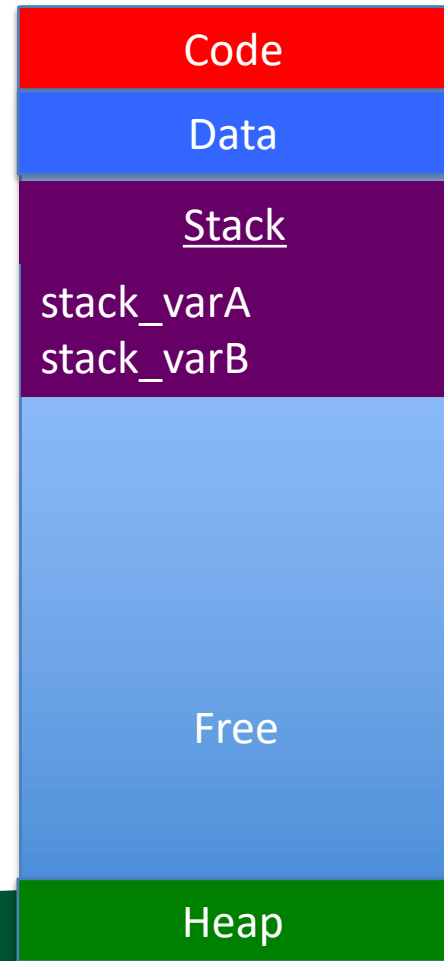
```
int main()  
{  
    int stack_varA; ←  
    {  
        int stack_varB = 3;  
    }  
}
```





Nested Scope

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



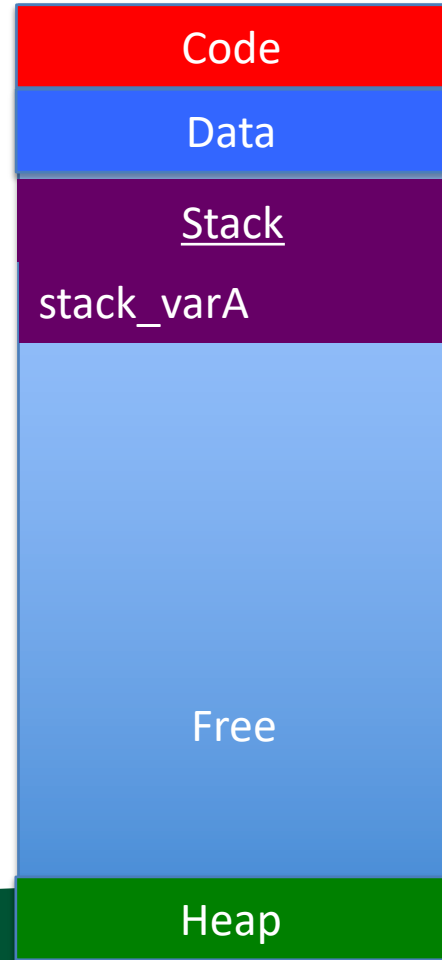


Nested Scope

```
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/Deallocation 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/Deallocation	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".

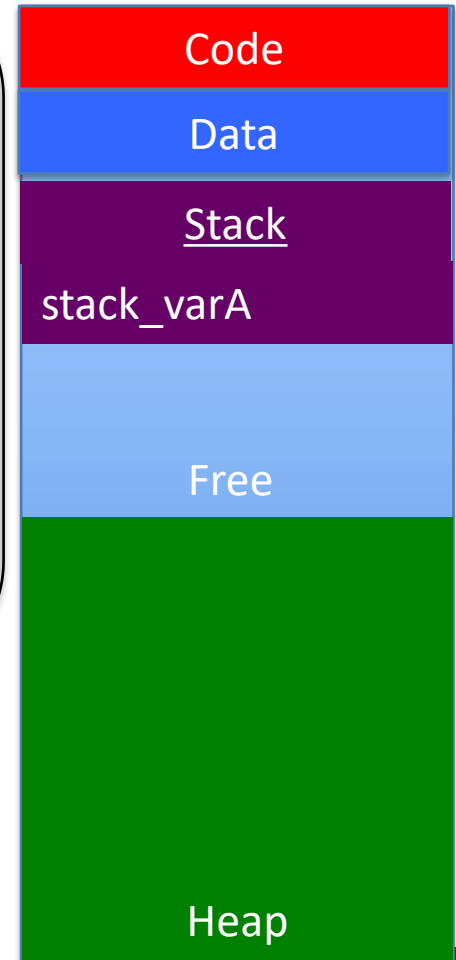
Memory leaks

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.

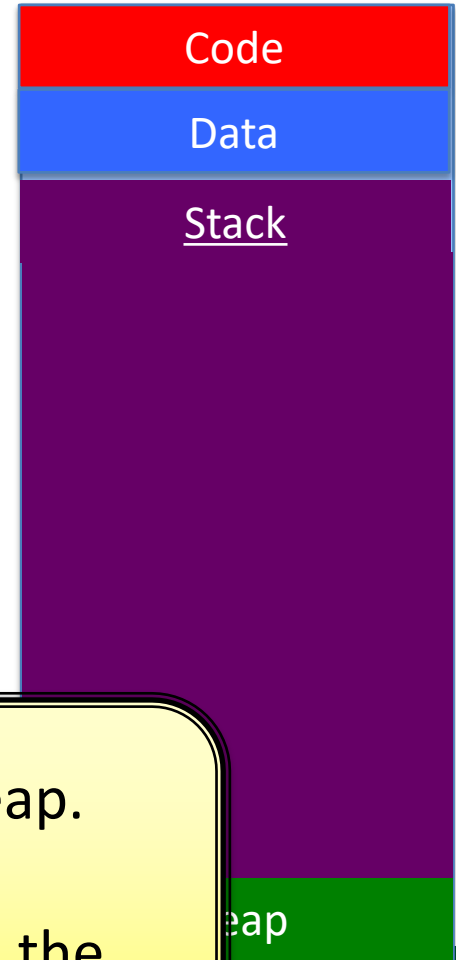
```
int i;  
int stack_varA;  
for (i = 0 ; i < 50 ; i++)  
    stack_varA = bar();  
}
```





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.)

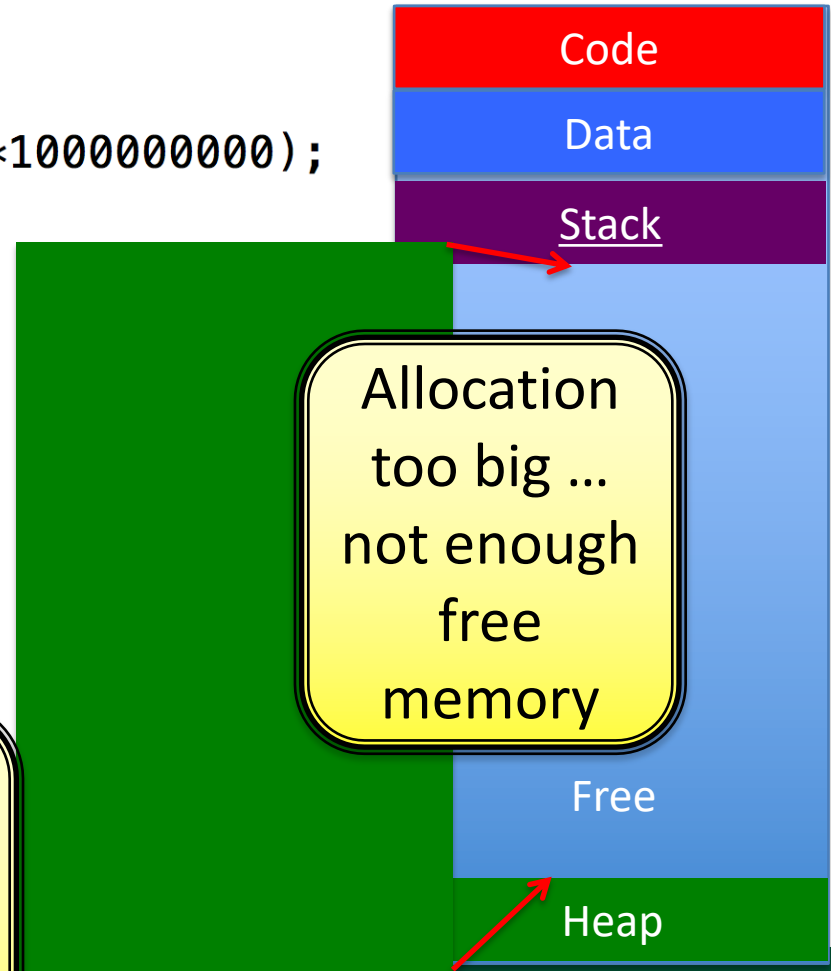


Running out of memory (heap)

```
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/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

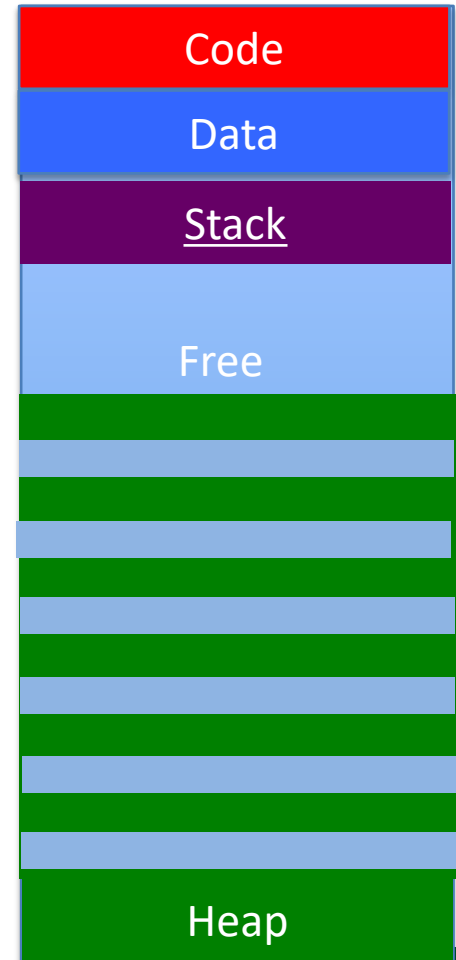
```
int *bar()
{
    int *heap_varA;
    heap_varA = malloc(sizeof(int)*2);
    heap_varA[0] = 2;
    heap_varA[1] = 2;
    return heap_varA;
}

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 fragmentation?

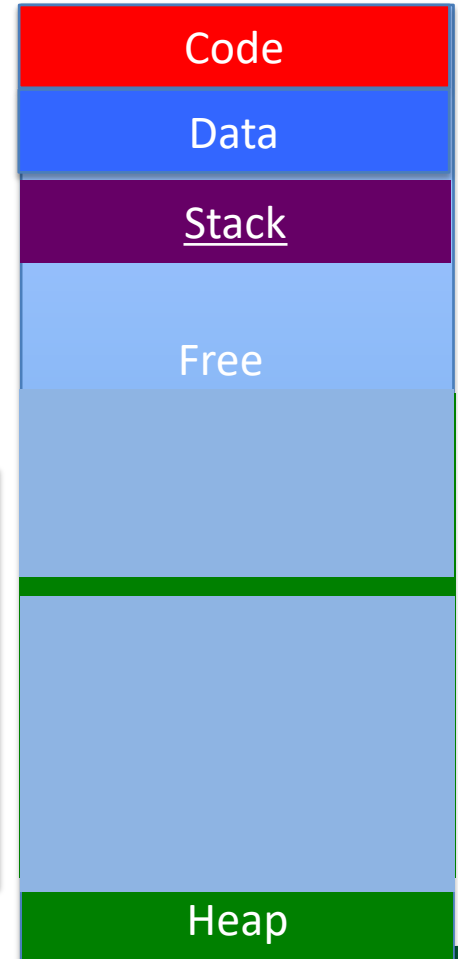
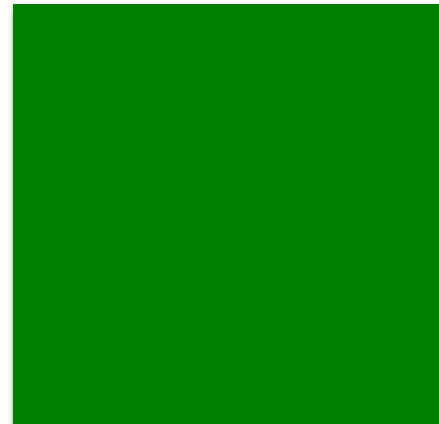
- (1) can't make big allocations
- (2) losing cache coherency





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/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

- Uninitialized memory read

```
int main()  
{  
    int *arr = malloc(sizeof(int)*10);  
    int v2=arr[3];  
}
```

When does this happen in real-world scenarios?