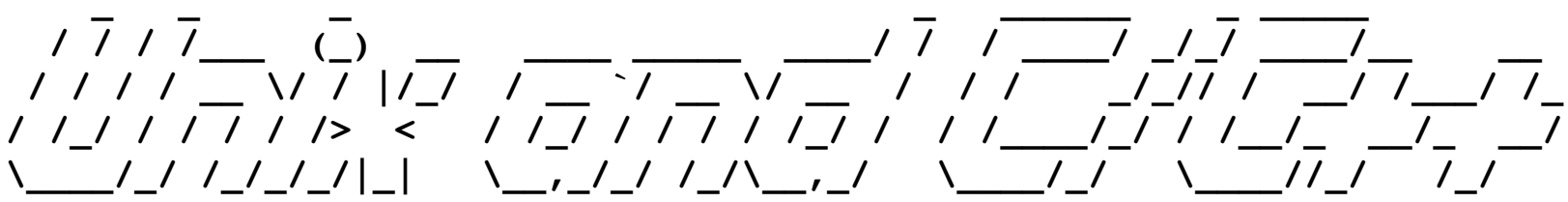
UNIVERSITY OF OREGON

CIS 330:



Lecture 4:
Build Systems, Tar, Character Strings

Lecture 5:
Finish up memory overview

April 9th/11th, 2018

Hank Childs, University of Oregon

(grade 2A)

Reading

- 2.6.1: environment variables
  - Does some $PATH stuff, I,.e, “./a.out”
- 2.6.2, 2.6.3: skip these for now
  - But it does talk about “pipes” (denoted as ‘|’)
  - In 2.7, when you see:
    - “tar -cvf CaDS.tar CaDS | column”
    - Just ignore the “| column” part
- 2.7: tar
- 4.1.2: make

Review

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)

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

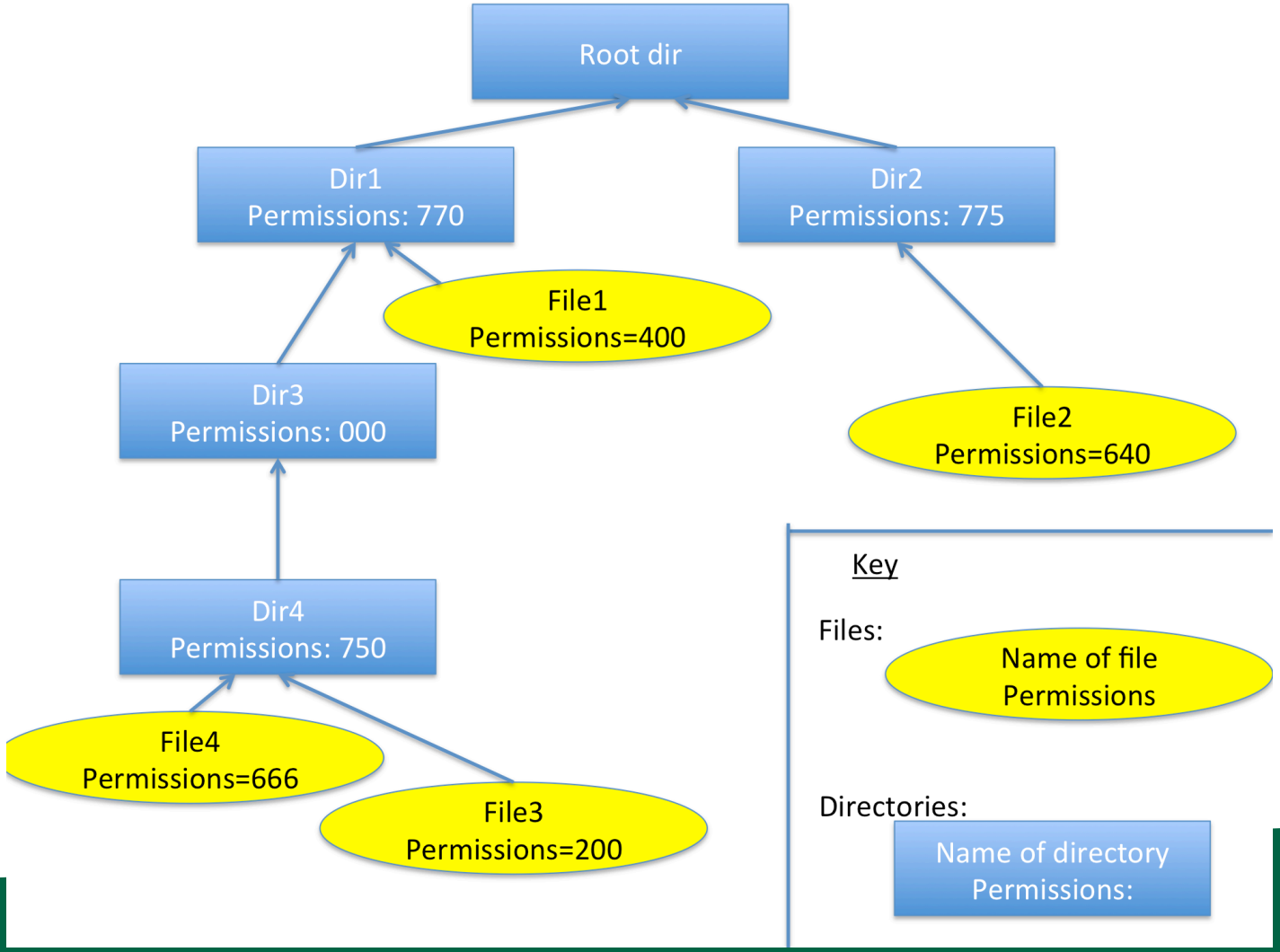
CIS 330: Project #1B
Assigned: April 6th, 2018
Due April 11th, 2018
(which means submitted by 6am on April 12th, 2018)
Worth 2% of your grade

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:

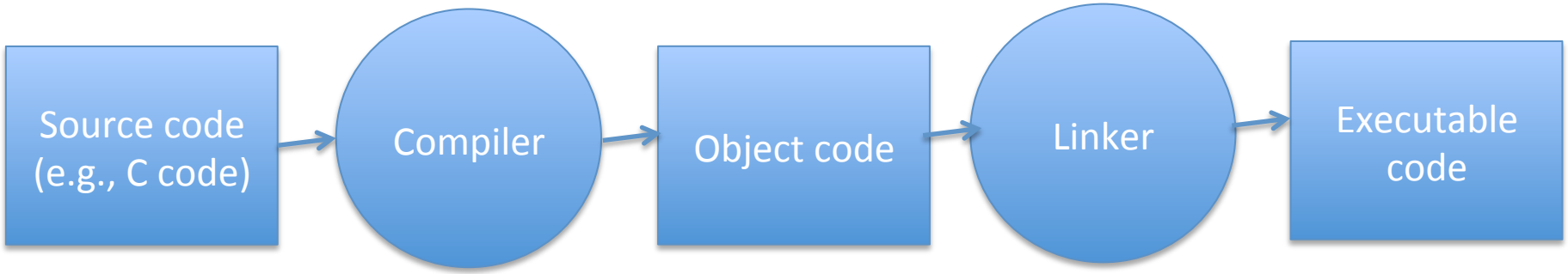


Outline

- Review
- Project 1B Overview
- Build
- Project 1C Overview
- Tar
- Character Strings

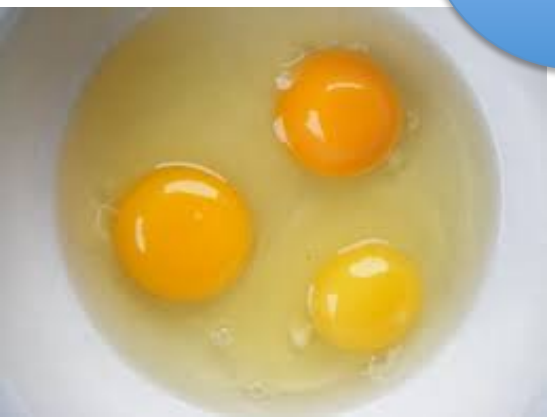
Build: The Actors

- File types
  - Source code
  - Object code
  - Executable code
- Programs
  - Compiler
  - Linker



Analogy

Source Code → Compiler → Object Code → Linker → Executable Code



Compilers, Object Code, and Linkers

- Compilers transform source code to object code
  - Confusing: most compilers also secretly have access to linkers and apply the linker for you.
- Object code: statements in machine code
  - not executable
  - intended to be part of a program
- Linker: turns object code into executable programs

GNU Compilers

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

C++ is superset of C.
With very few exceptions, every C program should compile with a C++ compiler.

C++ comments

- "//" : everything following on this line is a comment and should be ignored
- Examples:

```
// we set pi below
float pi = 3.14159; // approximation of pi
```

Can you think of a valid C syntax that will not compile in C++?

```
float radians=degrees//*approx. of pi*/3.14159;
```

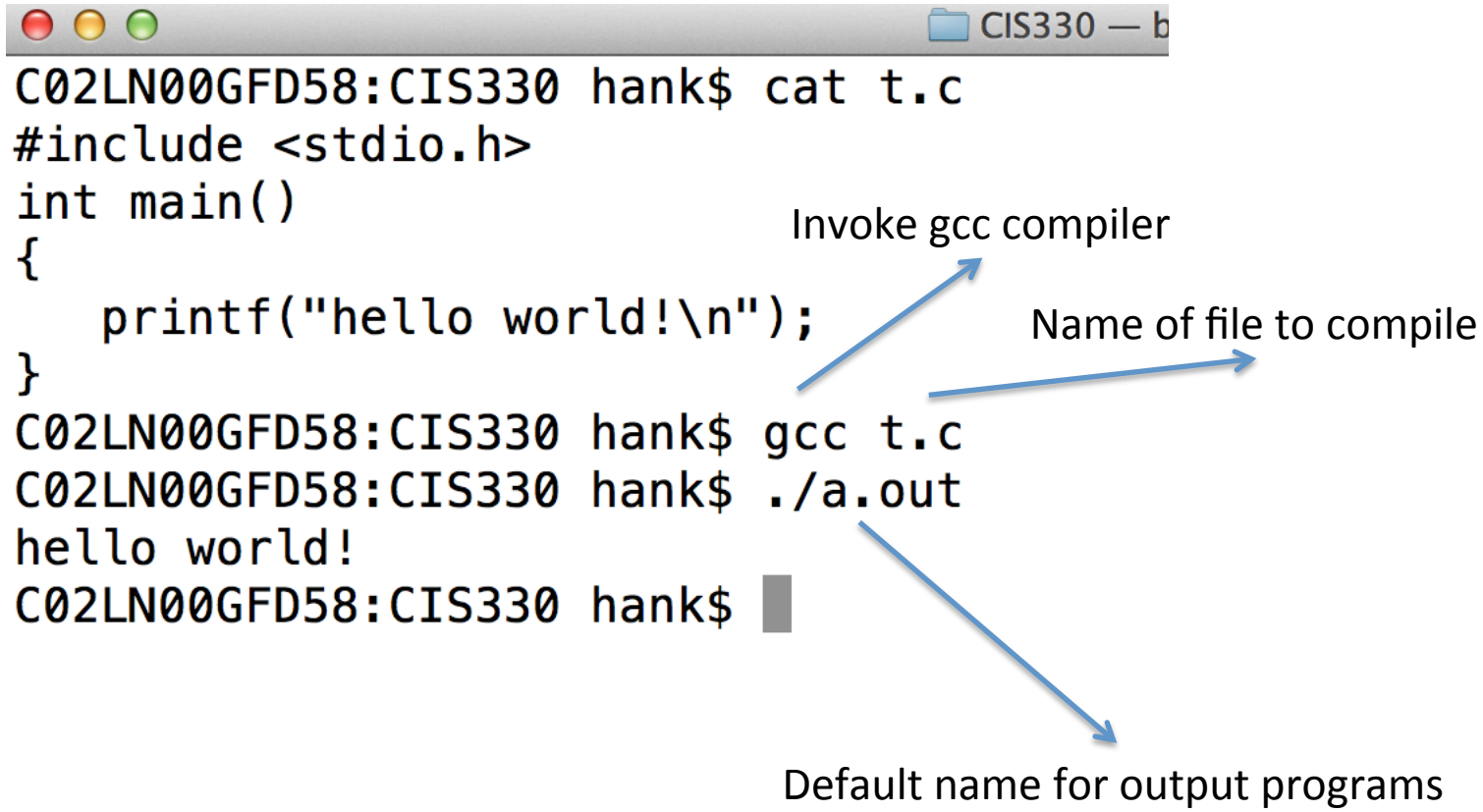
A comment on case (i.e., uppercase vs lowercase)

- Case is important in Unix
  - But Mac is tolerant
- gcc t.c

  →invokes C compiler
- gcc t.C

  → invokes C++ compiler

Our first gcc program



Our first gcc program: named output

CIS330 — bash — 80×24

```
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$ gcc -o helloworld t.c
C02LN00GFD58:CIS330 hank$ ./helloworld
hello world!
C02LN00GFD58:CIS330 hank$ ls -l helloworld
-rwxr-xr-x  1 hank  staff  8496 Apr  3 15:15 helloworld
C02LN00GFD58:CIS330 hank$
```

"-o" sets name of output

Output name is different

Output has execute permissions

gcc flags: debug and optimization

- "gcc –g": debug symbols
  - Debug symbols place information in the object files so that debuggers (gdb) can:
    - set breakpoints
    - provide context information when there is a crash
- "gcc –O2": optimization
  - Add optimizations … never fails
- "gcc –O3": provide more optimizations
  - Add optimizations … shouldn't fail, but can make "undefined behavior" worse and no longer has to mimic your code
- "gcc –O3 –g"
  - This is fine, but −g may bloat executables (possible slowdown)

Debug Symbols

- live code

```
int main()
{
    int sum = 0;
    int i;

    for (i = 0 ; i < 10 ; i++)
        sum += i;
    return sum;
}
```

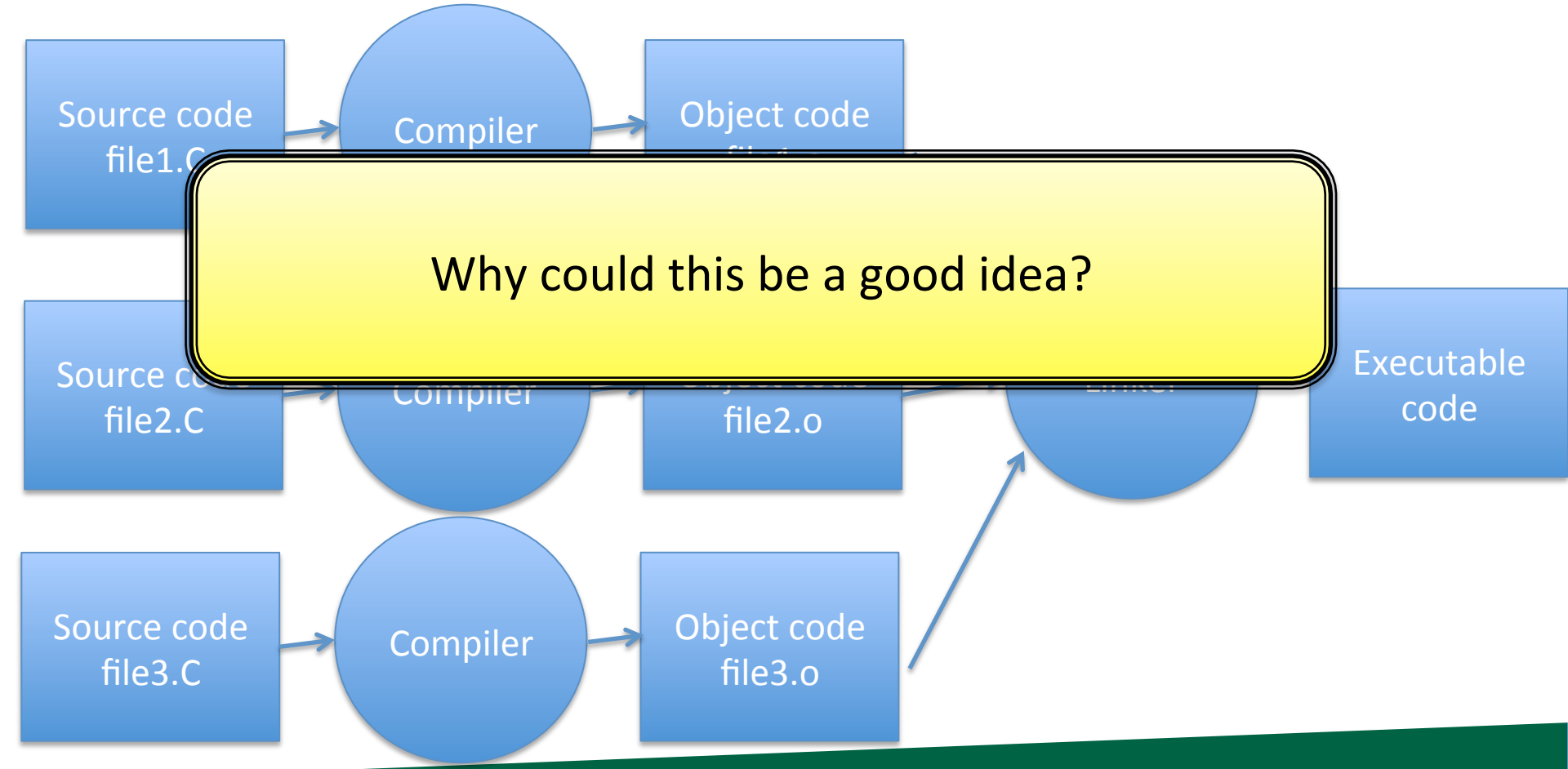
- gcc –S t.c # look at t.s
- gcc –S –g t.c # look at t.s

- (-S flag: compile to assembly instead of object code)

Object Code Symbols

- Symbols associate names with variables and functions in object code.
- Necessary for:
  - debugging
  - large programs

Large code development



Multi-file development: example

```
fawcett:330 childs$ cat t1.c
int doubler(int x)
{
   return 2*x;
}
fawcett:330 childs$ cat t2.c
int main()
{
    return doubler(5);
}
fawcett:330 childs$ gcc -c t1.c
fawcett:330 childs$ gcc -c t2.c
fawcett:330 childs$ gcc -o both t2.o t1.o
fawcett:330 childs$ ./both
fawcett:330 childs$ echo $?
10
```

cat is a Unix command that prints the contents of a file

Next slide has written description

$? is a shell construct that has the return value of the last executed program

How To Interpret Previous Slide

- gcc –c t1.c
  - → use the gcc compiler to create the object code file t1.o from the source code file t1.c
  - → "-c" is the flag that tells gcc that it should create just object code, and not try to call a linker
- gcc –c t2.c
  - Same as above
- gcc –o both t2.o t1.o
  - → use the gcc compiler to invoke a linker that will take the object code files t2.o and t1.o and combine them to make the executable code "both"

Multi-file development: example

```
fawcett:330 childs$ cat t1.c
int doubler(int x)
{
   return 2*x;
}
fawcett:330 childs$ cat t2.c
int main()
{
    return doubler(5);
}
fawcett:330 childs$ gcc -c t1.c
fawcett:330 childs$ gcc -c t2.c
fawcett:330 childs$ gcc -o both t2.o t1.o
fawcett:330 childs$ ./both
fawcett:330 childs$ echo $?
10
```

```
fawcett:330 childs$ gcc -o both t2.o
Undefined symbols:
  "_doubler", referenced from:
      _main in t2.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
fawcett:330 childs$ gcc -o both t1.o
Undefined symbols:
  "_main", referenced from:
     start in crt1.10.6.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

Multi-file development: example

```
fawcett:330 childs$ cat t1.c
int doubler(int x)
{
    return 2*x;
}
fawcett:330 childs$ cat t2.c
int main()
{
    return doubler(5);
}
fawcett:330 childs$ gcc -c t1
fawcett:330 childs$ gcc -c t2
fawcett:330 childs$ gcc -o bo
fawcett:330 childs$ ./both
fawcett:330 childs$ echo $?
10
```

```
fawcett:330 childs$ gcc -o both t1.o t2.o
fawcett:330 childs$
```

Linker order matters for some linkers (not Macs). Some linkers need the .o with "main" first and then extract the symbols they need as they go.
Other linkers make multiple passes.

Libraries

- Library: collection of "implementations" (functions!) with a well defined interface
- Interface comes through "header" files.
- In C, header files contain function prototypes and variables.
  - Accessed through "#include <file.h>"

Libraries

- Why are libraries a good thing?
- Answers:
  - separation
    - I.e., divide and conquer
      - increases productivity
    - I.e., simplicity
    - I.e., prevents connections between modules that shouldn't exist
  - encapsulation (hides details of the implementation)
    - "A little knowledge is a dangerous thing"…
    - Products
      - I can sell you a library and don't have to give you the source code.

Libraries

- Why are libraries a bad thing?
- Answers:
  - separation
    - I.e., makes connections between modules harder
      - (were the library interfaces chosen correctly?)
  - complexity
    - need to incorporate libraries into code compilation

Includes and Libraries

- gcc support for libraries
  - "-I": path to headers for library
    - when you say "#include <file.h>, then it looks for file.h in the directories -I points at
  - "-L": path to library location
  - "-lname": link in library libname

Library types

- Two types:
  - static and shared
- Static: all information is taken from library and put into final binary at link time.
  - library is never needed again
- Shared: at link time, library is checked for needed information.
  - library is loaded when program runs

More about shared and static later … for today, assume static

Making a static library

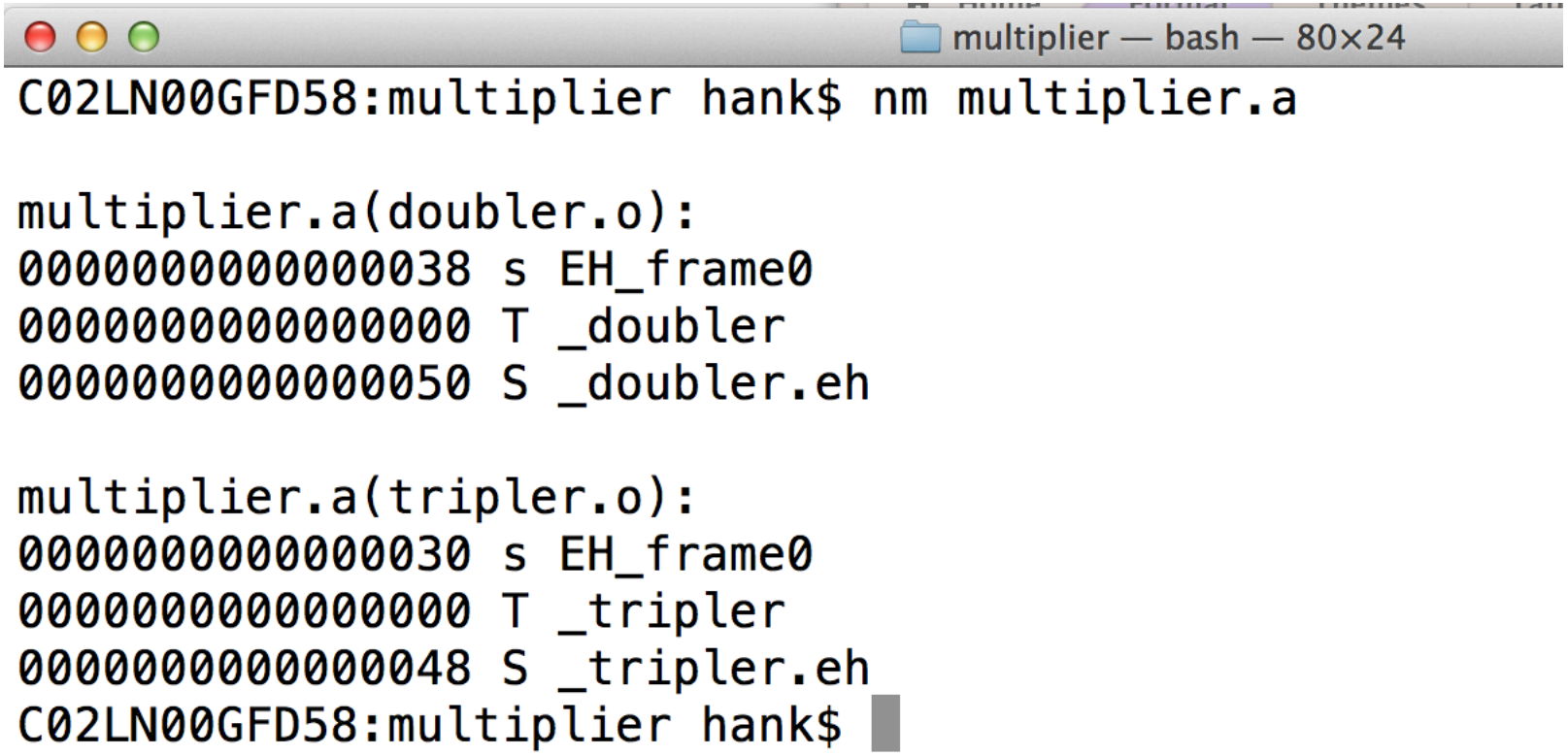
multiplier — bash — 80×24

```
C02LN00GFD58:multiplier hank$ cat multiplier.h # here's the header file
int doubler(int);
int tripler(int);
C02LN00GFD58:multiplier hank$ cat doubler.c # here's one of the c files
int doubler(int x) {return 2*x;}
C02LN00GFD58:multiplier hank$ cat tripler.c # here's the other c files
int tripler(int x) {return 3*x;}
C02LN00GFD58:multiplier hank$ gcc -c doubler.c # make an object file
C02LN00GFD58:multiplier hank$ ls doubler.o # we now have a .o
doubler.o
C02LN00GFD58:multiplier hank$ gcc -c tripler.c
C02LN00GFD58:multiplier hank$ ar r multiplier.a doubler.o tripler.o
C02LN00GFD58:multiplier hank$
```

(should have called this libmultiplier.a)

Note the '#' is the comment character

nm: What's in the file?



Typical library installations

- Convention
  - Header files are placed in “include” directory
  - Library files are placed in “lib” directory

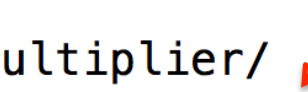
- Many standard libraries are installed in /usr
  - /usr/include
  - /usr/lib
- Compilers automatically look in /usr/include and /usr/lib (and other places)

Installing the library

multiplier — bash — 80×24

```
C02LN00GFD58:multiplier hank$ mkdir ~/multiplier
C02LN00GFD58:multiplier hank$ mkdir ~/multiplier/include
C02LN00GFD58:multiplier hank$ cp multiplier.h ~/multiplier/include/
C02LN00GFD58:multiplier hank$ mkdir ~/multiplier/lib
C02LN00GFD58:multiplier hank$ cp
doubler.c     multiplier.a  tripler.c
doubler.o     multiplier.h  tripler.o
C02LN00GFD58:multiplier hank$ cp multiplier.a ~/multiplier/
C02LN00GFD58:multiplier hank$ mv multiplier.a libmultiplier.a
C02LN00GFD58:multiplier hank$ cp libmultiplier.a ~/multiplier/lib/
C02LN00GFD58:multiplier hank$
```

(fixing my mistake)



"mv": unix command for renaming a file

Example: compiling with a library

CIS330 — bash — 80×24

```
C02LN00GFD58:CIS330 hank$ cat t.c
#include <multiplier.h>
#include <stdio.h>
int main()
{
   printf("Twice 6 is %d, triple 6 is %d\n", doubler(6), tripler(6));
}
C02LN00GFD58:CIS330 hank$ gcc -o mult_example t.c -I/Users/hank/multiplier/inclu
de -L/Users/hank/multiplier/lib -lmultiplier
C02LN00GFD58:CIS330 hank$ ./mult_example
Twice 6 is 12, triple 6 is 18
C02LN00GFD58:CIS330 hank$
```

- gcc support for libraries
  - "-I": path to headers for library
  - "-L": path to library location
  - "-lname": link in library libname

Makefiles

- There is a Unix command called "make"
- make takes an input file called a "Makefile"
- A Makefile allows you to specify rules
  - "if timestamp of A, B, or C is newer than D, then carry out this action" (to make a new version of D)
- make's functionality is broader than just compiling things, but it is mostly used for compilation

Basic idea: all details for compilation are captured in a file
… you just invoke "make" from a shell

Makefiles

- Reasons Makefiles are great:
  - Difficult to type all the compilation commands at a prompt
  - Typical develop cycle requires frequent compilation
  - When sharing code, an expert developer can encapsulate the details of the compilation, and a new developer doesn't need to know the details … just "make"

Makefile syntax

- Makefiles are set up as a series of rules
- Rules have the format:

  target: dependencies

  [tab] system command

Makefile example: multiplier lib

code — bash — 80×26

```
C02LN00GFD58:code hank$ cat Makefile
lib: doubler.o tripler.o
        ar r libmultiplier.a doubler.o tripler.o
        cp libmultiplier.a ~/multiplier/lib
        cp multiplier.h ~/multiplier/include

doubler.o: doubler.c
        gcc -c doubler.c

tripler.o: tripler.c
        gcc -c tripler.c
C02LN00GFD58:code hank$
C02LN00GFD58:code hank$
C02LN00GFD58:code hank$ make
ar r libmultiplier.a doubler.o tripler.o
cp libmultiplier.a ~/multiplier/lib
cp multiplier.h ~/multiplier/include
C02LN00GFD58:code hank$
C02LN00GFD58:code hank$ touch doubler.c
C02LN00GFD58:code hank$ make
gcc -c doubler.c
ar r libmultiplier.a doubler.o tripler.o
cp libmultiplier.a ~/multiplier/lib
cp multiplier.h ~/multiplier/include
C02LN00GFD58:code hank$
```

Fancy makefile example: multiplier lib

code — bash — 80×26

```
C02LN00GFD58:code hank$ cat Makefile
CC=gcc
CFLAGS=-g
INSTALL_DIR=~/multiplier

AR=ar
AR_FLAGS=r

SOURCES=doubler.c tripler.c
OBJECTS=$(SOURCES:.c=.o)

lib: $(OBJECTS)
        $(AR) $(AR_FLAGS) libmultiplier.a $(OBJECTS)
        cp libmultiplier.a $(INSTALL_DIR)/lib
        cp multiplier.h $(INSTALL_DIR)/include

.c.o:
        $(CC) $(CFLAGS) -c $<
C02LN00GFD58:code hank$ touch doubler.c
C02LN00GFD58:code hank$ make
gcc -g -c doubler.c
ar r libmultiplier.a doubler.o tripler.o
cp libmultiplier.a ~/multiplier/lib
cp multiplier.h ~/multiplier/include
C02LN00GFD58:code hank$
```

Configuration management tools

- Problem:
  - Unix platforms vary
    - Where is libX installed?
    - Is OpenGL supported?
- Idea:
  - Write program that answers these questions, then adapts build system
    - Example: put "-L/path/to/libX –lX" in the link line
    - Other fixes as well

Two popular configuration management tools

- Autoconf
  - Unix-based
  - Game plan:
    - You write scripts to test availability on system
    - Generates Makefiles based on results
- Cmake
  - Unix and Windows
  - Game plan:
    - You write .cmake files that test for package locations
    - Generates Makefiles based on results

CMake has been gaining momentum in recent years, because it is one of the best solutions for cross-platform support.

Outline

- Review
- Project 1B Overview
- Build
- Project 1C Overview
- Tar
- Character Strings

CIS 330: Project #1C
Assigned: April 7th, 2016
Due April 12th, 2016
(which means submitted by 6am on April 13th, 2016)
Worth 2% of your grade

Assignment: Download the file "Proj1C.tar". This file contains a C-based project. You will build a Makefile for the project, and also extend the project.

Project 1C

== Build a Makefile for math330 ==

Your Makefile should:

(1) create an include directory
(2) copy the Header file to the include directory
(3) create a lib directory
(4) compile the .c files in trig and exp as object files (.o's)
(5) make a library
(6) install the library to the lib directory
(7) compile the "cli" program against the include and library directory

== Extend the math330 library ==

You should:

(1) add 3 new functions: arccos, arcsin, and arctan (each in their own file)
(2) Extend the "cli" program to support these functions
(3) Extend your Makefile to support the new functions

Outline

- Review
- Project 1B Overview
- Build
- Project 1C Overview
- Tar
- Character Strings

Unix command: tar

- Anyone know what tar stands for?

tar = tape archiver

IBM tape library

Unix command: tar

- Problem: you have many files and you want to…
  - move them to another machine
  - give a copy to a friend
  - etc.
- Tar: take many files and make one file
  - Originally so one file can be written to tape drive
- Serves same purpose as “.zip” files.

Unix command: tar

- tar cvf 330.tar file1 file2 file3
  - puts 3 files (file1, file2, file3) into a new file called 330.tar
- scp 330.tar @ix:~ # Discussed Friday lab
- ssh ix # Discussed Friday lab
- tar xvf 330.tar
- ls

  file1 file2 file

SO SO SO IMPORTANT

```
Hanks-iMac:CIS330_S18 hank$ vi my_very_important_code.C
Hanks-iMac:CIS330_S18 hank$ # write code for hours
Hanks-iMac:CIS330_S18 hank$ # good
Hanks-iMac:CIS330_S18 hank$ tar cvf handin.tar my_ver_important_code.C f1.C f2.C f.3C # and more
Hanks-iMac:CIS330_S18 hank$ # very very bad
Hanks-iMac:CIS330_S18 hank$ tar cvf my_very_important_code.C f1.C f2.C f.3C handin.tar
```

Outline

- Review
- Project 1B Overview
- Build
- Project 1C Overview
- Tar
- Character Strings

ASCII Character Set

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

There have been various extensions to ASCII …
now more than 128 characters
Many special characters are handled outside this convention

image source: granneman.com

signed vs unsigned chars

- signed char (“char”):
  - valid values: -128 to 127
  - size: 1 byte
  - used to represent characters with ASCII
    - values -128 to -1 are not valid
- unsigned char:
  - valid values: 0 to 255
  - size: 1 byte
  - used to represent data

character strings

- A character “string” is:
  - an array of type “char”
  - that is terminated by the NULL character
- Example:

  char str[12] = “hello world”;
  - str[11] = ‘\0’ (the compiler did this automatically)
- The C library has multiple functions for handling strings

Character strings example

```
128-223-223-72-wireless:330 hank$ cat string.c
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char *str2 = str+6;

    printf("str is \"%s\" and str2 is \"%s\"\n",
           str, str2);

    str[5] = '\0';

    printf("Now str is \"%s\" and str2 is \"%s\"\n",
           str, str2);
}
128-223-223-72-wireless:330 hank$ gcc string.c
128-223-223-72-wireless:330 hank$ ./a.out
str is "hello world" and str2 is "world"
Now str is "hello" and str2 is "world"
```

Useful C library string functions

- strcpy: string copy
- strncpy: string copy, but just first N characters
- strlen: length of a string

```
128-223-223-72-wireless:330 hank$ cat strcpy.c
#include <string.h>
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char str2[6], str3[7];
    strcpy(str2, str+strlen("hello "));
    strncpy(str3, str, strlen("hello "));
    printf("%s,%s\n", str2, str3);
}
128-223-223-72-wireless:330 hank$ gcc strcpy.c
128-223-223-72-wireless:330 hank$ ./a.out
world,hello
```

Useful C library string functions

- strcpy: string copy
- strncpy: string copy, but just first N characters
- strlen: length of a string

```
128-223-223-72-wireless:330 hank$ cat strcpy.c
#include <string.h>
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char str2[7], str3[6];
    strcpy(str2, str+strlen("hello "));
    strncpy(str3, str, strlen("hello "));
    printf("%s,%s\n", str2, str3);
}
128-223-223-72-wireless:330 hank$ gcc strcpy.c
128-223-223-72-wireless:330 hank$ ./a.out
world,hello world
```

What happened here?

More useful C library string functions

Functions

Copying:

memcpy	Copy block of memory (function)
memmove	Move block of memory (function)
strcpy	Copy string (function)
strncpy	Copy characters from string (function)

Concatenation:

strcat	Concatenate strings (function)
strncat	Append characters from string (function)

Comparison:

memcmp	Compare two blocks of memory (function
strcmp	Compare two strings (function)
strcoll	Compare two strings using locale (functior
strncmp	Compare characters of two strings (functi
strxfrm	Transform string using locale (function)

Searching:

memchr	Locate character in block of memory (function)
strchr	Locate first occurrence of character in string (fu
strcspn	Get span until character in string (function)
strpbrk	Locate characters in string (function)
strrchr	Locate last occurrence of character in string (fu
strspn	Get span of character set in string (function)
strstr	Locate substring (function)
strtok	Split string into tokens (function)

Other:

memset	Fill block of memory (function)
strerror	Get pointer to error message string (function)
strlen	Get string length (function)

Macros

NULL	Null pointer (macro)

Types

size_t	Unsigned integral type (type)

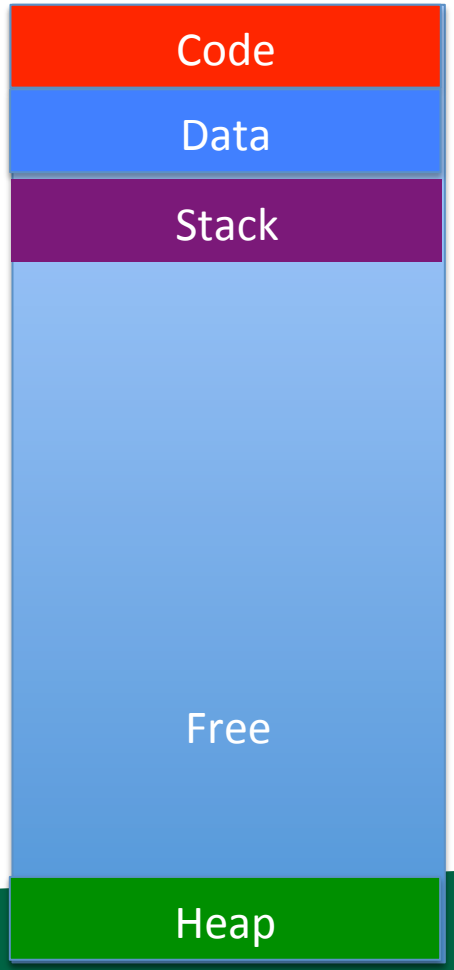
source: cplusplus.com

Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/ Deallocation	Automatic	Explicit

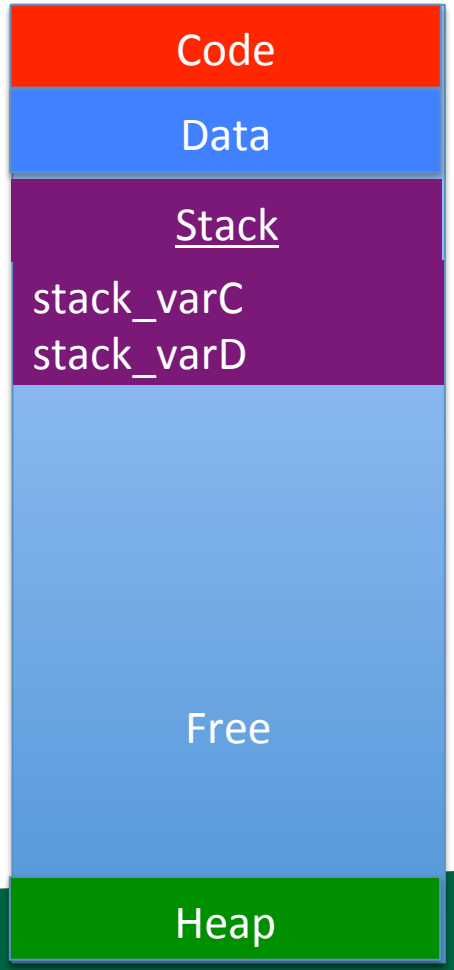
How stack memory is allocated into Stack Memory Segment

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

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



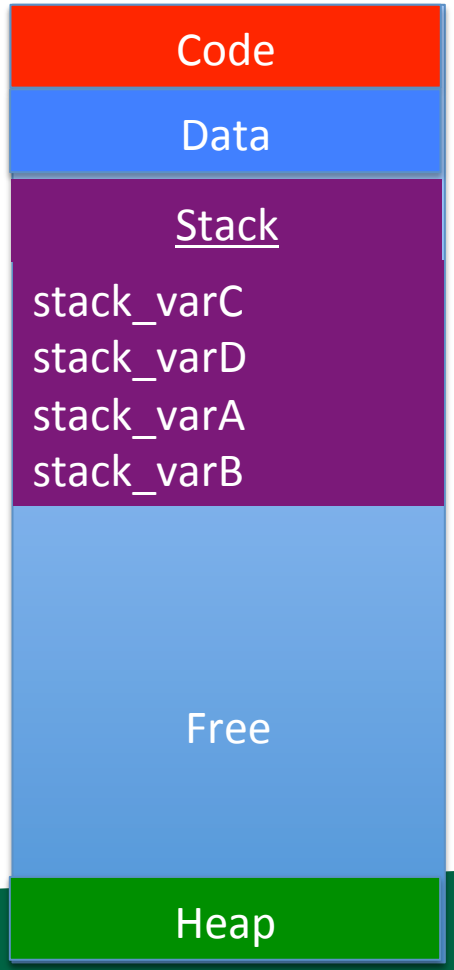
How stack memory is allocated into Stack Memory Segment

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

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



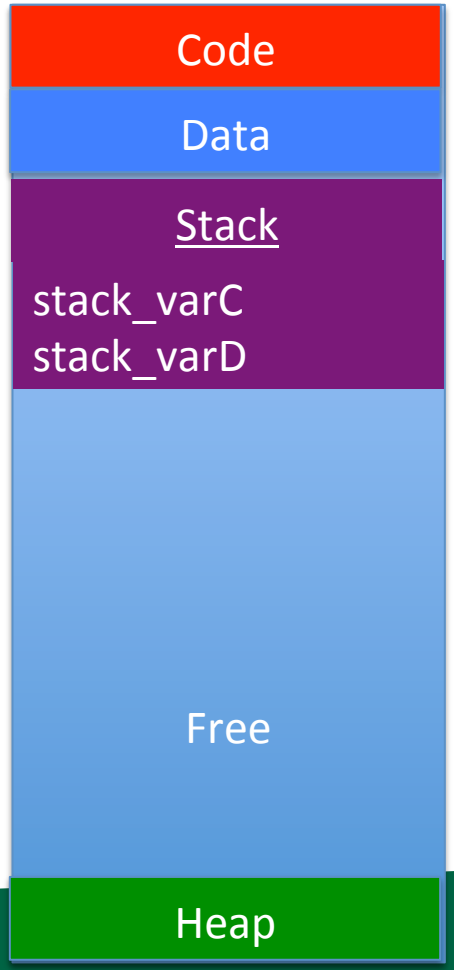
How stack memory is allocated into Stack Memory Segment

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

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



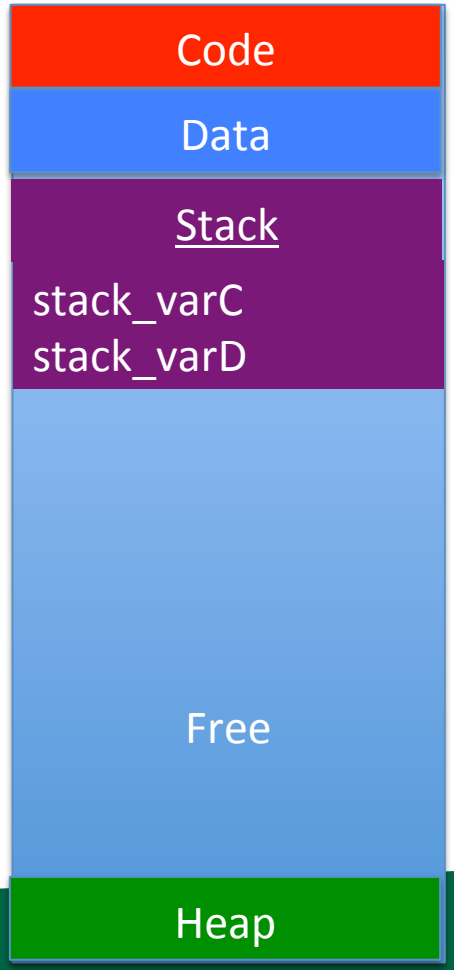
How stack memory is allocated into Stack Memory Segment

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

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



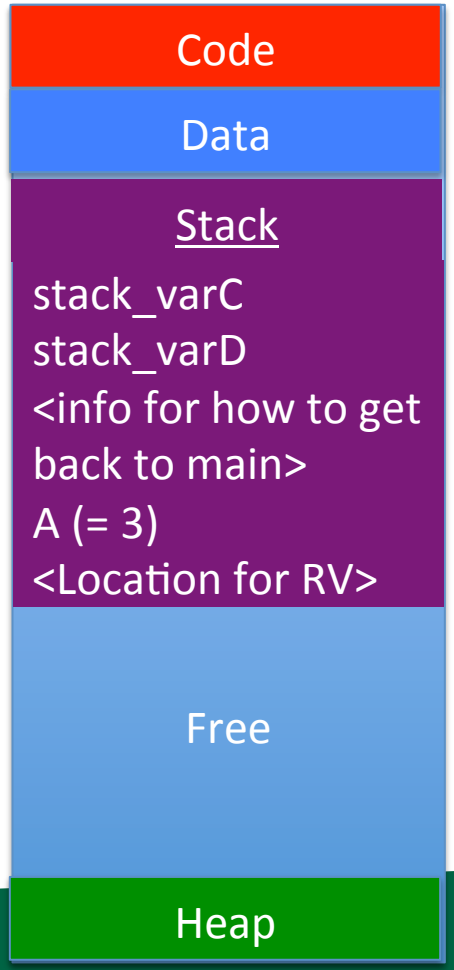
How stack memory is allocated into Stack Memory Segment

```
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);
}
```



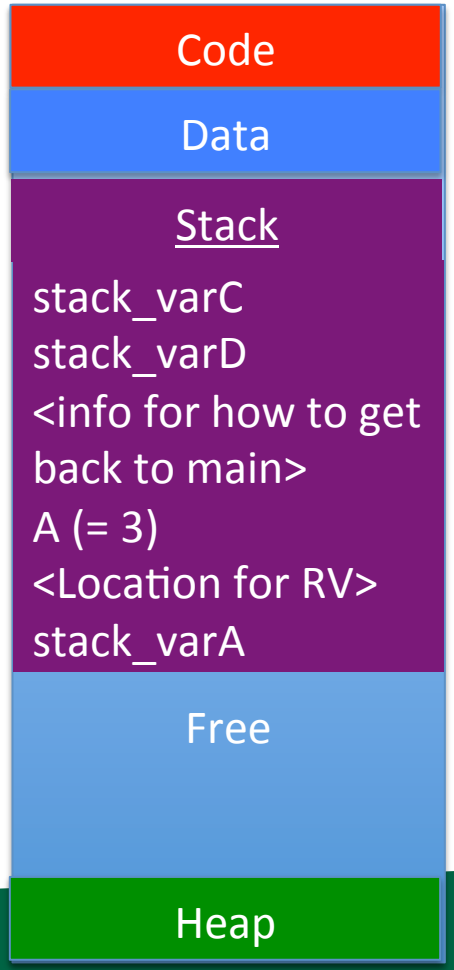
How stack memory is allocated into Stack Memory Segment

```
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

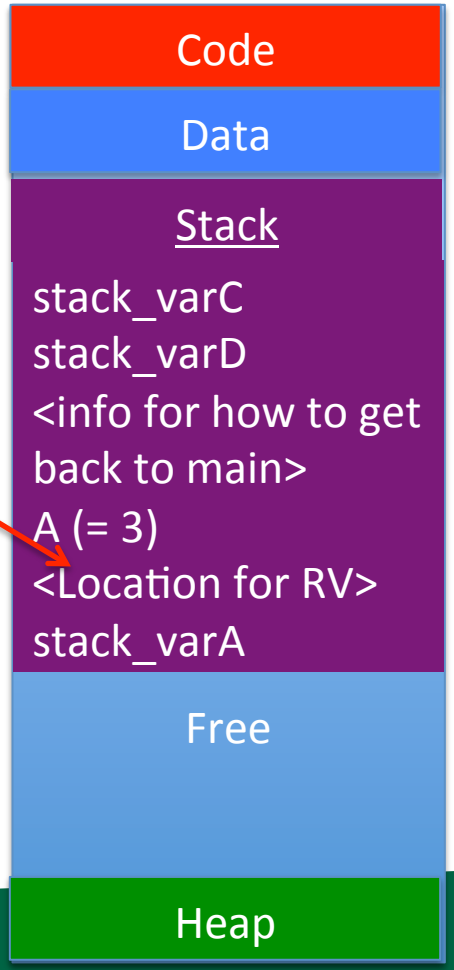
```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

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



How stack memory is allocated into Stack Memory Segment

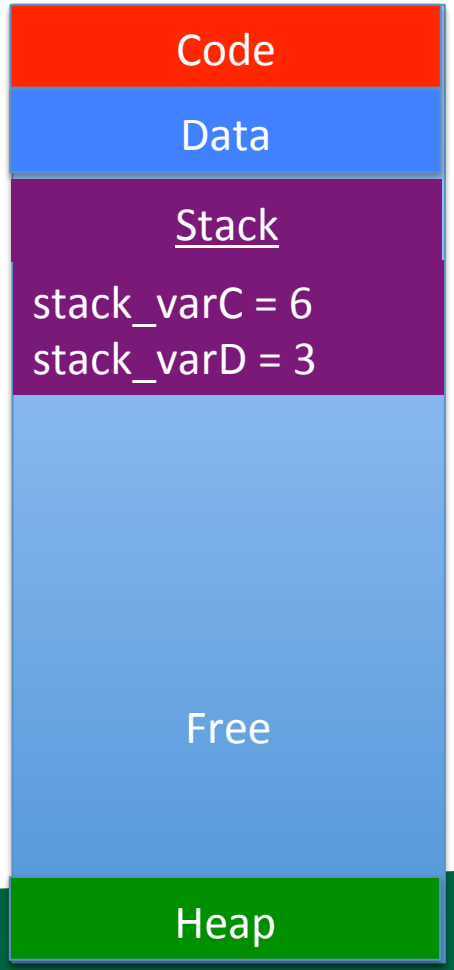
```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

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);
}
```



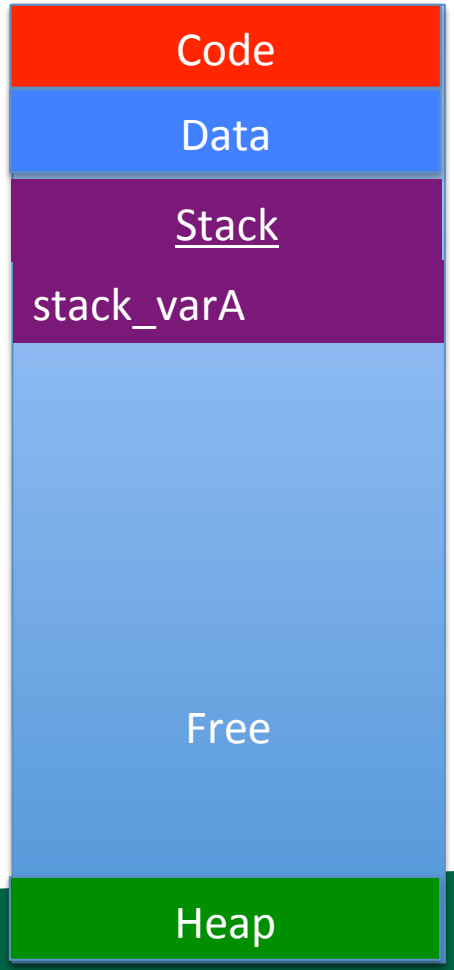
This code is very problematic … why?

```
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;
    }
}
```



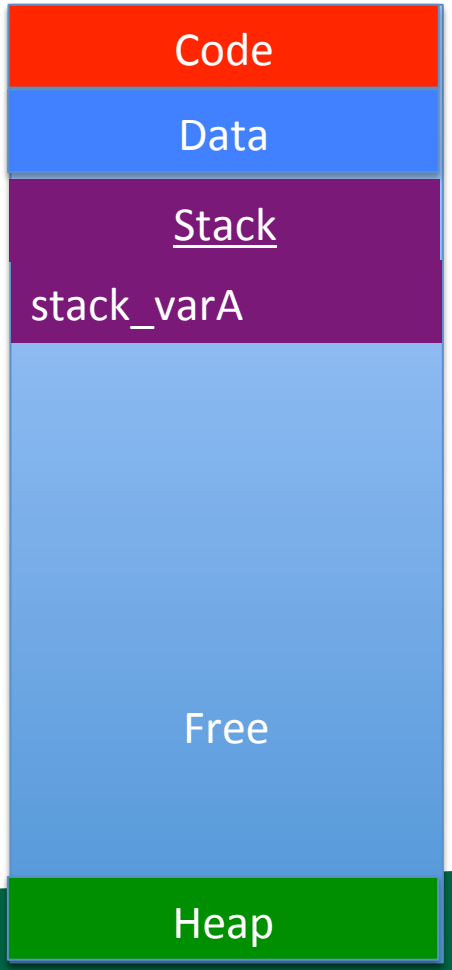
Nested Scope

```
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".

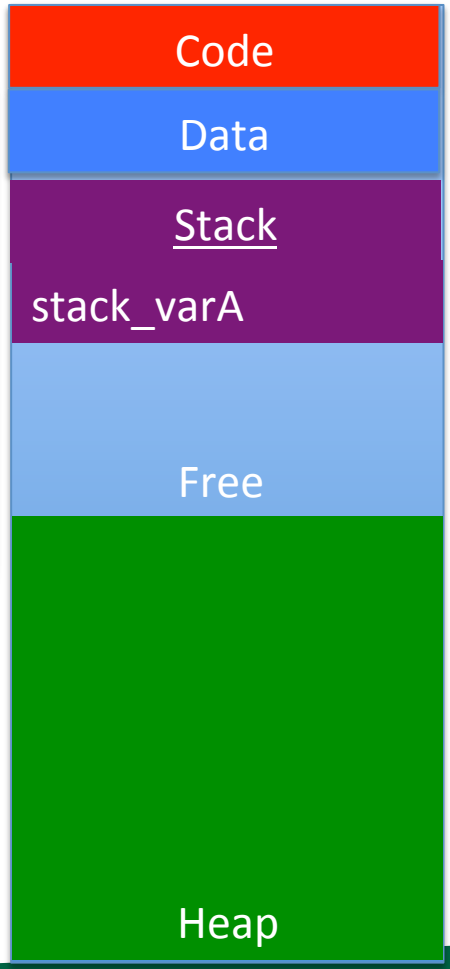
Memory leaks

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

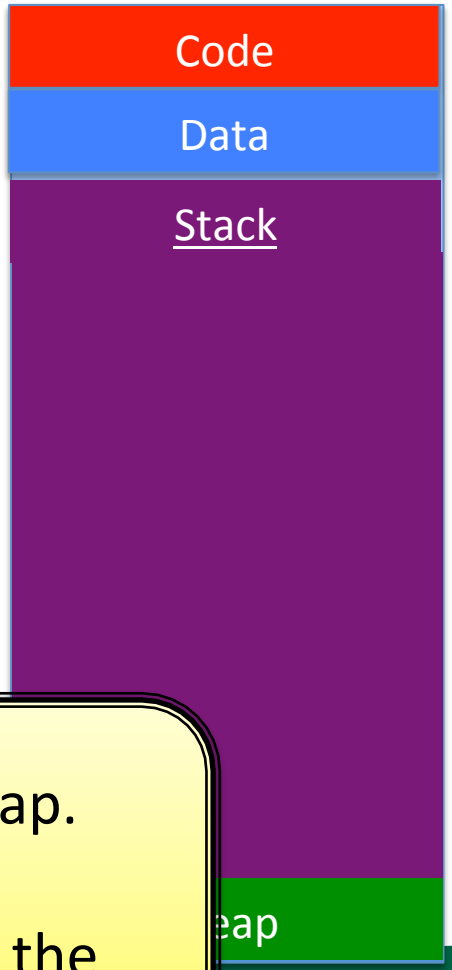
The heap's memory manager will not be able to re-claim them … we have effectively limited the memory available to the program.

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



Running out of memory (stack)

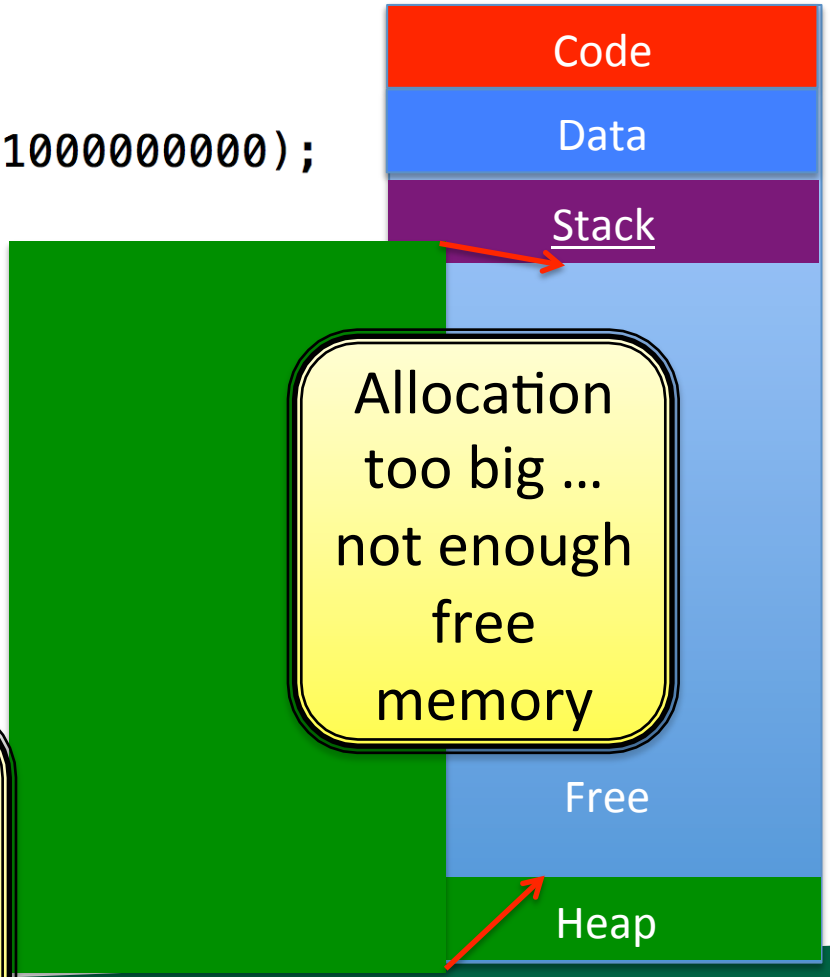
```
int endless_fun()
{
    endless_fun();
}

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



stack overflow: when the stack runs into the heap.
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]);
}
```

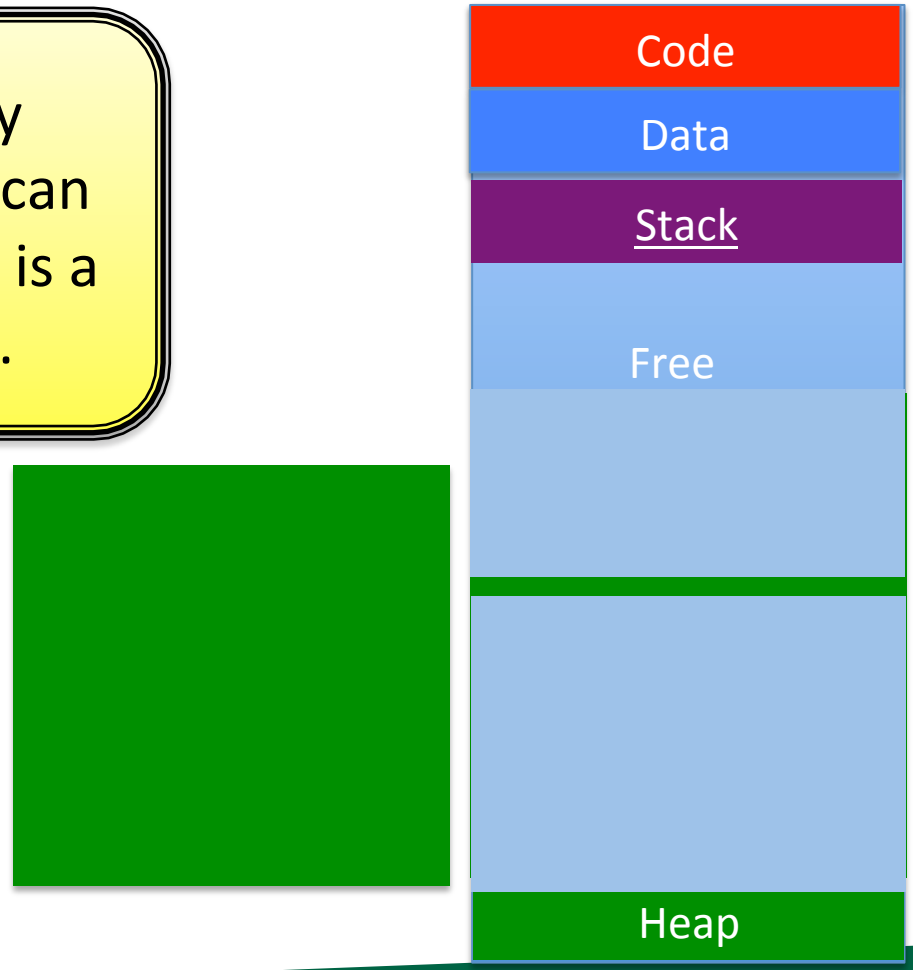
Negative aspects of fragmentation?

(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];
}
```

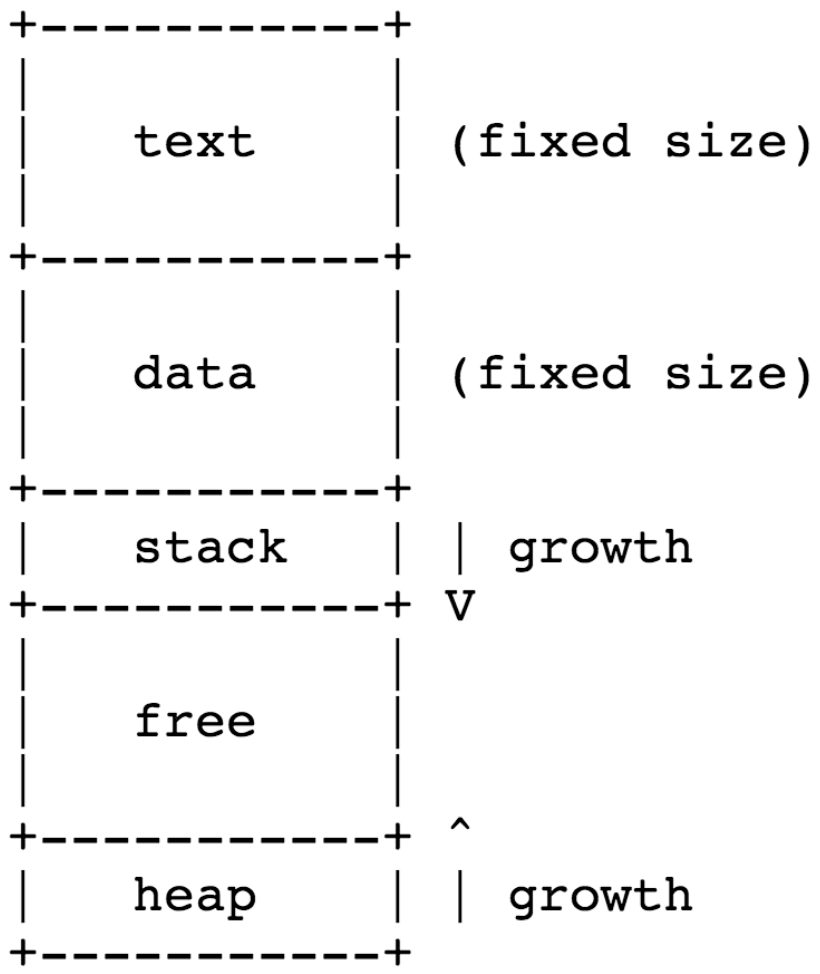
- Array bounds write

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

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

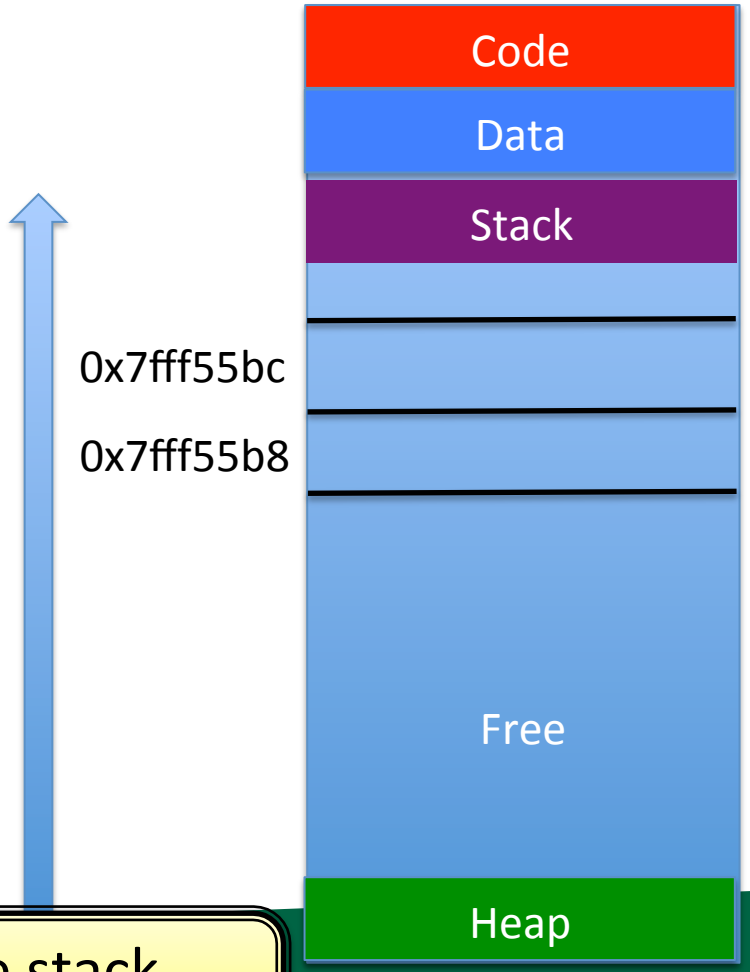
- sizeof: gets size of type
- Usually:
  - sizeof(int) == 4
  - sizeof(float) == 4
  - sizeof(double) == 8
  - sizeof(unsigned char) == 1
  - sizeof(char) == 1
  - sizeof(int *) == sizeof(double *) == sizeof(char *) == 8
- → array of 10 ints → malloc(10*sizeof(int))

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;
```

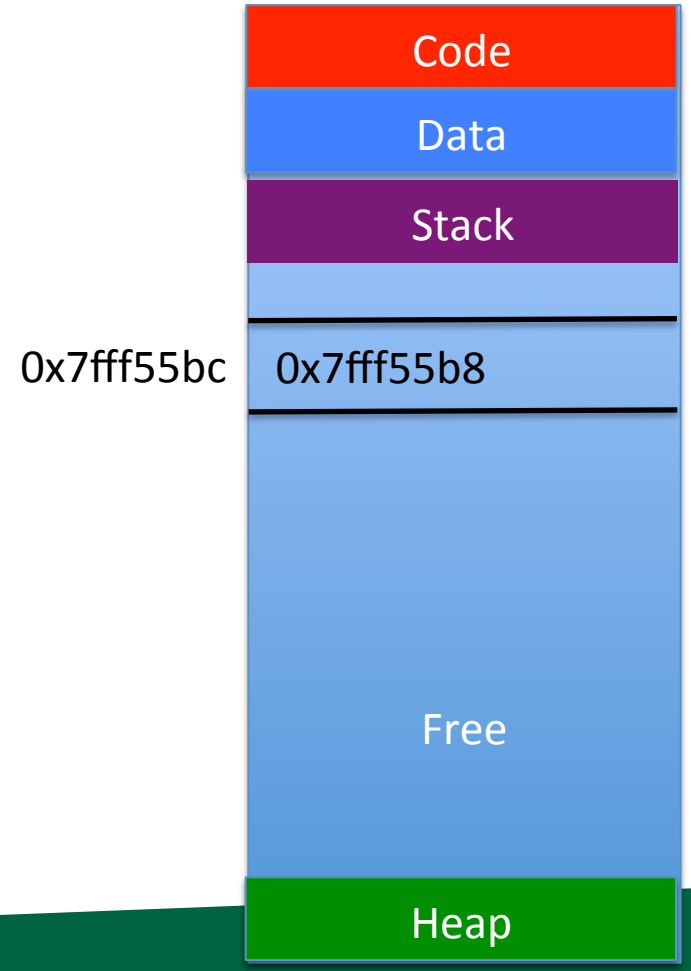
B=(A+5) → A[5] = B[0]

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

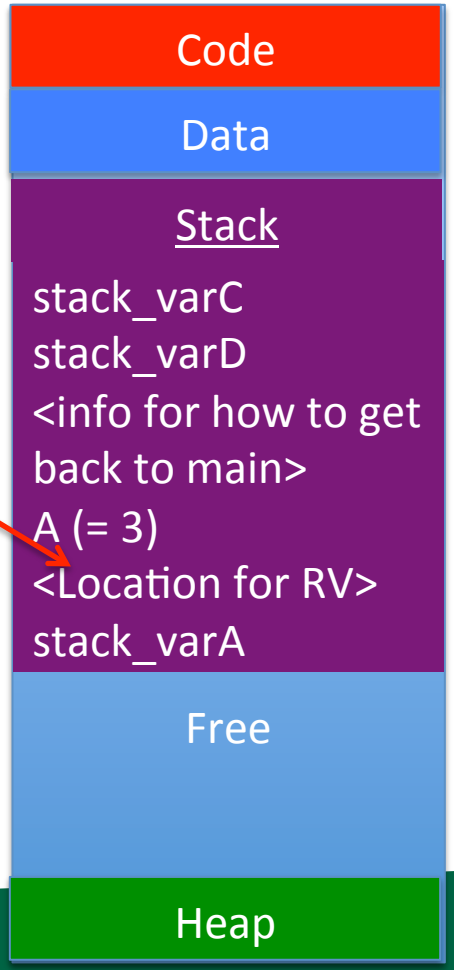
Call by Value

```
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



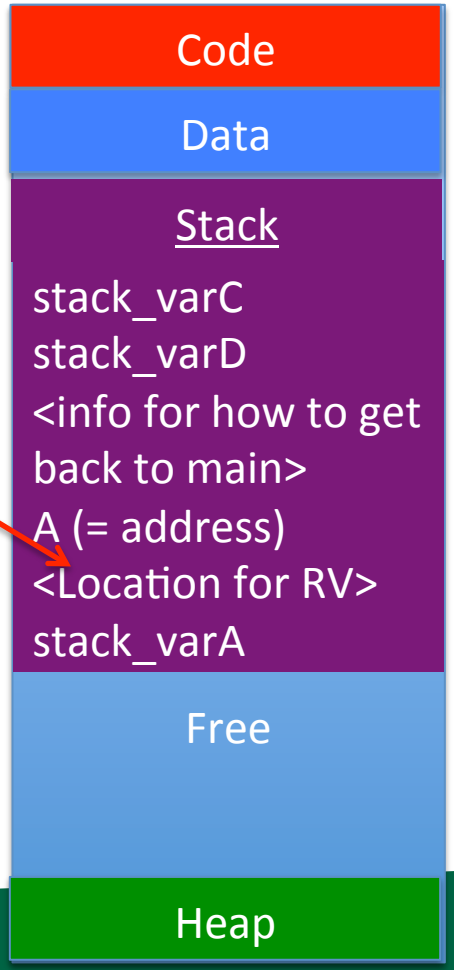
Call by reference

```
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?

Misc. Stuff for 4A

memcpy

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

NAME
     memcpy -- copy memory area

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <string.h>

     void *
     memcpy(void *restrict dst, const void *restrict src, size_t n);

DESCRIPTION
     The memcpy() function copies n bytes from memory area src to memory area
     dst.  If dst and src overlap, behavior is undefined.  Applications in
     which dst and src might overlap should use memmove(3) instead.

RETURN VALUES
     The memcpy() function returns the original value of dst.
```

I mostly use C++, and I still use memcpy all the time

sscanf

- like printf, but it parses from a string

  sscanf(str, "%s\n%d %d\n%d\n", magicNum,
  &width, &height, &maxval);

  on:

  str="P6\n1000 1000\n255\n";

  gives:

  magicNum = "P6", width = 1000,

  height = 1000, maxval = 255

if-then-else

```
int val = (X < 2 ? X : 2);
```

←→

```
if (X<2)
{
   val = X;
}
else
{
   val = 2;
}
```