Lecture 2:
Memory in C
Office Hours

• Hank’s OH:
  – Tues 2pm-3pm, Friday, 11am-noon

• Hank’s OH Location: 301 Deschutes Hall

• Brent’s Office Hours:
  – Monday: 4-5pm
  – Weds: 4-5pm (not today)
  – Thurs: 4-5pm (not today)

• TA OH Location: 100 Deschutes Hall
Note on Homeworks

• Project 1A: assigned Monday, due tonight
  – → means 6am Thursday
  – Will discuss again in 10 slides

• Project 2A: assigned today, due in class on Monday

• Project 1B assigned Friday, due Weds Apr 12

• Lecture this Friday (not lab)
VirtualBox Demo

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

**WHEN**
Fri, April 06
4:00 - 6:00 PM

**WHERE**
PSC Boo6 (Vislab)

https://library.uoregon.edu/sclib/psc-dearmond-makerspace
More on Piazza
Plan for today

• Quick review of Unix basics
• Project 1A
• Baby steps into C and gcc
• Memory
Plan for today

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

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

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

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

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

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

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

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

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

• rmdir: removes a directory
  – Two flavors
    • Relative to current directory
      – rmdir badDir
    • Relative to absolute path
      – rmdir /dir1/dir2/badDir
        » Removes badDir, leaves dir1, dir2 in place

• Only works on empty directories!
  – “Empty” directories are directories with no files

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

• touch: “touch” a file

• Behavior:
  – If the file doesn’t exist
    • → create it
  – If the file does exist
    • → update time stamp

Time stamps record the last modification to a file or directory

Will talk more about this command with build systems
Unix commands: ls

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

• Get a man page:

• → “man rmdir” gives:

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

NAME
   rmdir -- remove directories

SYNOPSIS
   rmdir [-p] directory ...

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

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

   The following option is available:

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

• vmtutor a great start for learning “vi”
• But ask me for tips any time you see me editing
# vi / vim graphical cheat sheet

### Esc (normal mode)

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>toggle case</td>
</tr>
<tr>
<td>.</td>
<td>ggoto mark</td>
</tr>
<tr>
<td>1</td>
<td>external filter</td>
</tr>
<tr>
<td>2</td>
<td>@ play macro</td>
</tr>
<tr>
<td>3</td>
<td># prev ident</td>
</tr>
<tr>
<td>4</td>
<td>$ eol</td>
</tr>
<tr>
<td>5</td>
<td>% goto match</td>
</tr>
<tr>
<td>6</td>
<td>^ &quot;soft&quot; bol</td>
</tr>
<tr>
<td>7</td>
<td>&amp; repeat :s</td>
</tr>
<tr>
<td>8</td>
<td>* next ident</td>
</tr>
<tr>
<td>9</td>
<td>( begin sentence</td>
</tr>
<tr>
<td>0</td>
<td>) &quot;hard&quot; bol</td>
</tr>
<tr>
<td>-</td>
<td>prev line</td>
</tr>
<tr>
<td>+</td>
<td>next line</td>
</tr>
<tr>
<td>=</td>
<td>= auto-format</td>
</tr>
</tbody>
</table>

### Q (ex mode)

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>record macro</td>
</tr>
<tr>
<td>w</td>
<td>next word</td>
</tr>
<tr>
<td>e</td>
<td>end word</td>
</tr>
<tr>
<td>r</td>
<td>replace char</td>
</tr>
<tr>
<td>t</td>
<td>back 'till</td>
</tr>
<tr>
<td>y</td>
<td>yank line</td>
</tr>
<tr>
<td>u</td>
<td>undo</td>
</tr>
<tr>
<td>i</td>
<td>insert mode</td>
</tr>
<tr>
<td>o</td>
<td>open above</td>
</tr>
<tr>
<td>p</td>
<td>paste before</td>
</tr>
<tr>
<td>{</td>
<td>begin parag.</td>
</tr>
<tr>
<td>}</td>
<td>end parag.</td>
</tr>
</tbody>
</table>

### A (append)

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>append at eol</td>
</tr>
<tr>
<td>s</td>
<td>subst line</td>
</tr>
<tr>
<td>d</td>
<td>delete to eol</td>
</tr>
<tr>
<td>f</td>
<td>&quot;find&quot; ch</td>
</tr>
<tr>
<td>g</td>
<td>extra cmd</td>
</tr>
<tr>
<td>h</td>
<td>screen top</td>
</tr>
<tr>
<td>j</td>
<td>join lines</td>
</tr>
<tr>
<td>k</td>
<td>help</td>
</tr>
<tr>
<td>l</td>
<td>screen bottom</td>
</tr>
<tr>
<td>x</td>
<td>ex cmd line</td>
</tr>
<tr>
<td>:</td>
<td>&quot;reg. spec</td>
</tr>
</tbody>
</table>

### Z (quit)

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>quit</td>
</tr>
<tr>
<td>x</td>
<td>xtra cmd</td>
</tr>
</tbody>
</table>

### X (delete char)

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>xtra cmd</td>
</tr>
</tbody>
</table>

### C (change to eol)

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>change to eol</td>
</tr>
</tbody>
</table>

### V (visual mode)

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>visual mode</td>
</tr>
</tbody>
</table>

### B (prev word)

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>prev word</td>
</tr>
</tbody>
</table>

### N (next find)

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>next find</td>
</tr>
</tbody>
</table>

### M (set mark)

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>set mark</td>
</tr>
</tbody>
</table>

### Screen

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>reverse t/T/F/F</td>
</tr>
<tr>
<td>g</td>
<td>repeat t/T/F/F</td>
</tr>
<tr>
<td>4</td>
<td>&quot;goto mk. bol</td>
</tr>
</tbody>
</table>

### Not used!

- !
- 13
- 14
- 21
- 22
- 23

---

### Motion

- moves the cursor, or defines the range for an operator
- direct action command, if red, it enters insert mode
- requires a motion afterwards, operates between cursor & destination

### Command

- special functions, requires extra input commands with a dot need a char argument afterwards
- bol = beginning of line, eol = end of line, mk = mark, yank = copy

### Operator

- words: `quux` (foo, bar, baz)
- WORDS: `quux` (foo, bar, baz)

### Extra

- `q` key

### Main command line commands ('ex'):

- `:w` (save), `:q` (quit), `:q!` (quit w/o saving)
- `:e f` (open file f), `:e %s/x/y/g` (replace 'x' by 'y' filewide), `:i` (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 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](http://www.viemu.com) - home of ViEmu, vi/vim emulation for Microsoft Visual Studio
Plan for today

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

- Practice using an editor
- Must be written using editor on Unix platform
  - I realize this is unenforceable.
  - If you want to do it with another mechanism, I can’t stop you
    - But realize this project is simply to prepare you for later projects
Project 1A

• Write >=300 words using editor (vi, emacs, other)

• Topic: what you know about C programming language

• Can’t write 300 words?
  – Bonus topic: what you want from this course

• How will you know if it is 300 words?
  – Unix command: “wc” (word count)
Unix command: wc (word count)

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

(63 = lines, 252 = words, 1071 = character)
Project 1A

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

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

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

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

• Canvas

• If you run into trouble:
  – Email me your solution
Plan for today

• Quick review of Unix basics
• Project 1A
• Baby steps into C and gcc
• Memory
GNU Compilers

• GNU compilers: open source
  – gcc: GNU compiler for C
  – g++: GNU compiler for C++
Our first gcc program

```
#include <stdio.h>
int main()
{
    printf("hello world!\n");
}
```

Unix command that prints contents of a file

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

Invoke gcc compiler

```
C02LN00GFD58:CIS330 hank$ gcc t.c
C02LN00GFD58:CIS330 hank$ ./a.out
```

Default name for output programs

```
hello world!
C02LN00GFD58:CIS330 hank$
```

You should use this for Proj 2A.
Plan for today

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

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

- You can control the memory
- That helps get good performance
- If you don’t control the memory (like in other programming languages), you are likely to get poor performance
- ... so let’s talk about memory
**Motivation: Project 2A**

Assignment: fill out this worksheet.

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

Code:

```c
int *A = 0x8000;
```

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Your Answer</th>
<th>Variable</th>
<th>Your Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0x8000</td>
<td>(A+6)-(A+3)</td>
<td></td>
</tr>
<tr>
<td>&amp;A</td>
<td>NOT ENOUGH INFO</td>
<td><em>(A+6)-</em>(A+4)</td>
<td></td>
</tr>
<tr>
<td>*A</td>
<td></td>
<td>(A+6)-B[0]</td>
<td></td>
</tr>
</tbody>
</table>
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

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

    /* deallocates memory */
    free(ptr);
}
Important Memory Concepts in C (4/9): Arrays

• Arrays lie in contiguous memory
  – So if you know address to one element, you know address of the rest
• int *a = malloc(sizeof(int)*1);
  – a single integer
  – ... or an array of a single integer
• int *a = malloc(sizeof(int)*2);
  – an array of two integers
  – first integer is at ‘a’
  – second integer is at the address ‘a+4’
    • Tricky point here, since C/C++ will refer to it as ‘a+1’
Important Memory Concepts in C (5/9): Dereferencing

• There are two operators for getting the value at a memory location: *, and []
  – This is called deferencing
    • * = “dereference operator”

• int *p = malloc(sizeof(int)*1);
• *p = 2; /* sets memory p points to to have value 2 */
• p[0] = 2; /* sets memory p points to to have value 2 */
Important Memory Concepts in C (6/9): pointer arithmetic

- int *p = malloc(sizeof(int)*5);
- C/C++ allows you to modify pointer with math operations
  - called pointer arithmetic
  - “does the right thing” with respect to type
    - int *p = malloc(sizeof(int)*5);
    - p+1 is 4 bytes bigger than p!!

- Then:
  - “p+3” is the same as “&p[3]” (ADDRESSES)
  - “*(p+3)” is the same as “p[3]” (VALUES)
Important Memory Concepts in C (7/9)

Pointers to pointers

- `int **p = malloc(sizeof(int *)*5);`
- `p[0] = malloc(sizeof(int)*50);`
- ...
Important Memory Concepts in C (8/9): Hexadecimal address

- Addresses are in hexadecimal
- `int *A = 0x8000;`
- Then `A+1` is `0x8004`. (Since `int` is 4 bytes)
Important Memory Concepts in C (9/9)

NULL pointer

• int *p = NULL;
• often stored as address 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

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

• Assigned now

• Worksheet. You print it out, complete it on your own, and bring it to class.

• Due Monday 10am in class
  – Graded in class

• No Piazza posts on this please

• Practice with C, vi, gcc, printf
Memory Segments

• Von Neumann architecture: one memory space, for both instructions and data

• → so break memory into “segments”
  – ... creates boundaries to prevent confusion

• 4 segments:
  – Code segment
  – Data segment
  – Stack segment
  – Heap segment
Code Segment

• Contains assembly code instructions
• Also called text segment
• This segment is modify-able, but that’s a bad idea
  – “Self-modifying code”
    • Typically ends in a bad state very quickly.
Data Segment

• Contains data not associated with heap or stack
  – global variables
  – statics (to be discussed later)
  – character strings you’ve compiled in
    char *str = “hello world\n”
Stack: data structure for collection

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

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

• Stack: memory set aside as scratch space for program execution

• When a function has local variables, it uses this memory.
  – When you exit the function, the memory is lost
Stack

• The stack grows as you enter functions, and shrinks as you exit functions.
  – This can be done on a per variable basis, but the compiler typically does a grouping.
    • Some exceptions (discussed later)

• Don’t have to manage memory: allocated and freed automatically
Heap

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

- text (fixed size)
- data (fixed size)
- stack | growth
- free
- heap | growth

Source: http://www.cs.uwm.edu/classes/cs315/Bacon/
## Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th>Allocation/Deallocation</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
</tbody>
</table>
How stack memory is allocated into Stack Memory Segment

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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?

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

Nested Scope

```c
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    } // End of nested block
}
```

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

- Code
- Data
- Stack
  - stack_varA
- Free
- Heap
# Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocation/Deallocation</strong></td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td><strong>Access</strong></td>
<td>Fast</td>
<td>Slower</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocation/Deallocation</strong></td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td><strong>Access</strong></td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td><strong>Variable scope</strong></td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
</tbody>
</table>
Variable scope: stack

```c
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.

```c
{ 
  int i;
  int stack_varA;
  for (i = 0 ; i < 50 ; i++)
    stack_varA = bar();
}
```
Running out of memory (stack)

```c
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)

```c
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

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocation/Deallocation</strong></td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td><strong>Access</strong></td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td><strong>Variable scope</strong></td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
<tr>
<td><strong>Fragmentation</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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

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

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deallocation</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td>Access</td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td>Variable scope</td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Fragmentation</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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

```c
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

```c
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

```c
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
  ```c
  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

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