Lecture 3: Build Systems, Memory
Announcements

• Matt’s OH: Mon 12-1, Tues 12-2
• Hank’s OH: Weds 11-12, Fri 12:30-1:30
• It sounds like Weds 11-12 is problematic for many.
  – I am very constrained on Weds
  – Starting next week, I can offer an “on demand” OH on Weds after class.
Outline

• Review: permissions
• Review: project 1B
• Review: basics of build
• New: build
• New: project 1C
• New: memory allocation (maybe)
Outline

• Review: permissions
• Review: project 1B
• Review: basics of build
• New: build
• New: project 1C
• New: memory allocation (maybe)
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?

A bunch of bits ... we could represent this with binary

User = “owner”
Other = “not owner, not group”
Translating permissions to binary

<table>
<thead>
<tr>
<th>#</th>
<th>Permission</th>
<th>rwx</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>full</td>
<td>111</td>
</tr>
<tr>
<td>6</td>
<td>read and write</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>read and execute</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>read only</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>write and execute</td>
<td>011</td>
</tr>
<tr>
<td>2</td>
<td>write only</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>execute only</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>none</td>
<td>000</td>
</tr>
</tbody>
</table>

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)
Outline

• Review: permissions
• Review: project 1B
• Review: basics of build
• New: build
• New: memory allocation (maybe)
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!!
CIS 330: Project #1B  
Assigned: April 3rd, 2015  
Due April 8th, 2015  
(which means submitted by 6am on April 9th, 2014)  
Worth 2% of your grade

Assignment: Create a 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”. (A consistent name will help with grading.)

Note: you are only allowed to use the following commands: mkdir, touch, cd, chmod,  
mv, cp. (You do not need to use all of these commands to successfully complete the  
assignment.)
Project 1B

The directory structure should be:

```
Root dir

Dir1
  Permissions: 770

  File1
    Permissions: 400

Dir2
  Permissions: 775

  File2
    Permissions: 640

Dir3
  Permissions: 000

Dir4
  Permissions: 750

  File4
    Permissions: 666

  File3
    Permissions: 200
```

Key

Files:
- Name of file Permissions

Directories:
- Name of directory Permissions
Outline

• Review: permissions
• Review: project 1B
• Review: basics of build
• New: build
• New: memory allocation (maybe)
Build: The Actors

- File types
  - Source code
  - Object code
  - Executable code

- Programs
  - Compiler
  - Linker
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++
Our first gcc program

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

Invoke gcc compiler
Name of file to compile
Default name for output programs

Note: compiler is calling linker directly, object file is intermediate only and not stored to file system
Our first gcc program: named output

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

```
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
Outline

• Review: permissions
• Review: project 1B
• Review: basics of build
• New: build
• New: project 1C
• New: memory allocation (maybe)
Object Code Symbols

• Symbols associate names with variables and functions in object code.

• Necessary for:
  – debugging
  – large programs
Imagine a world without symbols...

• I make object file (.o) that is part of an executable
• It has a function called “foo”
• You make an object file containing function called “foo2” that calls “foo”
• ... and linker wants to make an executable with these two object files
• If there are no symbols, linker just has addresses to work with

Symbols provide hints to linker & debugger that allow them function
Do an example with nm to show symbols
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 ... sometimes fails

• “gcc –O3 –g”
  – Debugging symbols slow down execution ... and sometimes compiler won’t do it anyways...
Large code development

Why could this be a good idea?
gcc flag “-c”: make object code, but don’t link

• gcc –c file1.c
  – → makes object code file “file1.o”
Multi-file development: example

```c
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.

**$?** is a shell construct that has the return value of the last executed program.
Multi-file development: example

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

```c
int doubler(int x) {
    return 2*x;
}
```

```
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 t1.o t2.o
fawcett:330 childs$ ./both
fawcett:330 childs$ echo $? 10
```

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 functions 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 tendrils 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
  – “-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
ar: archiver

• Makes a library
  – i.e., collects binary code in object files into a single file (a library)

• Usage: ar libname.a file1.o file2.o
Making a static library

Note the ‘#’ is the comment character

```bash
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)
What’s in the file?

C02LN00GFD58:multiplier hank$ nm multiplier.a

multiplier.a(doubler.o):
000000000000000038 s EH_frame0
0000000000000000 T _doubler
00000000000000050 S _doubler.eh

multiplier.a(tripler.o):
000000000000000030 s EH_frame0
0000000000000000 T _tripler
00000000000000048 S _tripler.eh
C02LN00GFD58:multiplier hank$
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

```
C02LN00GF58:multiplier hank$ mkdir ~/multiplier
C02LN00GF58:multiplier hank$ mkdir ~/multiplier/include
C02LN00GF58:multiplier hank$ cp multiplier.h ~/multiplier/include/
C02LN00GF58:multiplier hank$ mkdir ~/multiplier/lib
C02LN00GF58:multiplier hank$ cp doubler.c multiplier.a tripler.c
doubler.o multiplier.h tripler.o
C02LN00GF58:multiplier hank$ cp multiplier.a ~/multiplier/
C02LN00GF58:multiplier hank$ mv multiplier.a libmultiplier.a
C02LN00GF58:multiplier hank$ cp libmultiplier.a ~/multiplier/lib/
C02LN00GF58:multiplier hank$
```

“mv”: unix command for renaming a file
Example: compiling with a library

```c
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));
}
```

```bash
C02LN00GFD58:CIS330 hank$ gcc -o mult_example t.c -I/Users/hank/multiplier/include -L/Users/hank/multiplier/lib -lmultiplier
C02LN00GFD58:CIS330 hank$ ./mult_example
Twice 6 is 12, triple 6 is 18
```

- 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 computation

Basic idea: all details for compilation are captured in a configuration 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

  Target: what to build. (a name you give to make.)
  Dependencies: what it depends on (files in the filesystem or other rules)
  System command: gcc ...
Makefile example: multiplier lib

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

stripler.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

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 problem 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: permissions
• Review: project 1B
• Review: basics of build
• New: build
• New: project 1C
• New: memory allocation (maybe)
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
- scp 330.tar @ix:~
- ssh ix
- tar xvf 330.tar
- ls
  
  file1 file2 file
Project 1C

CIS 330: Project #1C
Assigned: April 8th, 2015
Due April 12th, 2015
(which means submitted by 6am on April 13th, 2015)
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: permissions
• Review: project 1B
• Review: basics of build
• New: build
• New: project 1C
• New: memory allocation (yes!)
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></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deallocation</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
</tbody>
</table>


Allocation / Deallocation

```c
#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);
}
```
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

Stack
- stack_varC
- stack_varD
- <info for how to get back to main>
- A (= 3)
- <Location for RV>

Heap

Free
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 };  // this may be on the stack
    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 overwri-
(en and bar’s stack_varD overwrites foo’s stack_varC in this program)
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}

Nested Scope

Code
Data
Stack
stack_varA

Free
Heap
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}

```c
// Code
Code

// Data
Data

// Stack
Stack
stack_varA
stack_varB

// Free
Free

// Heap
Heap
```
Nested Scope

```c
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

<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>
</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>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>
</tbody>
</table>
Variable scope: stack and heap

```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 */
}
```

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 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
{i
  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>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 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

```c
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

<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>
Outline

• Announcements/Review
• Project 2B
• Project 2C
• Memory Overview
• Memory Errors
• Finish Unix Boot Camp
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];
}
```

When does this happen in real-world scenarios?