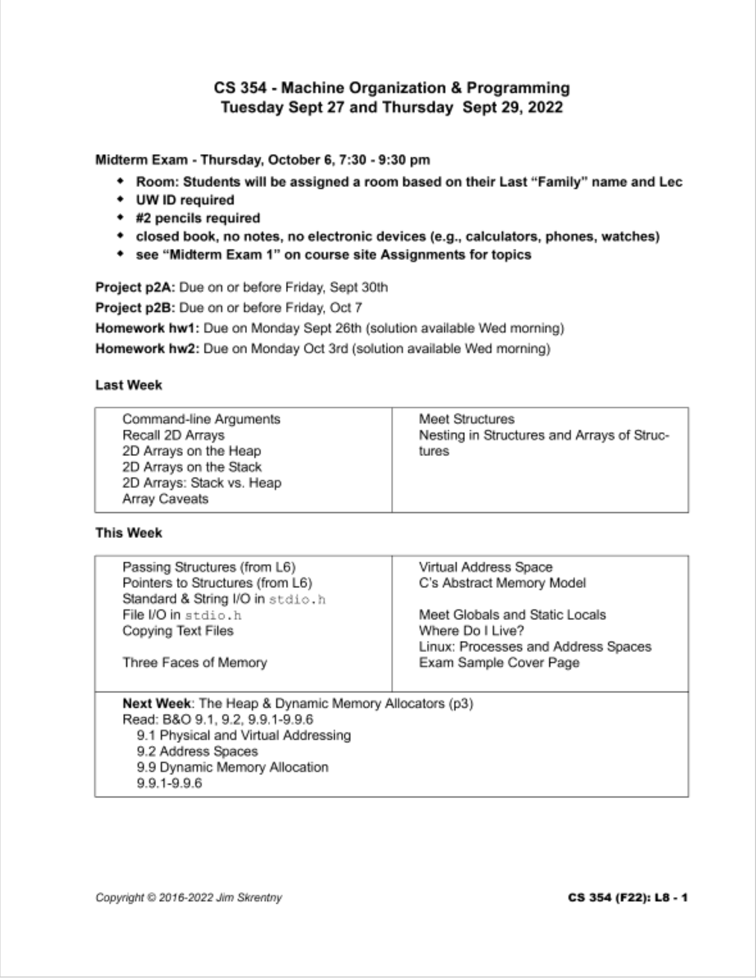
# outlineL08-w4TR-student

Thursday, October 6, 2022 2:41 PM

outlineL08-w4TR-student



## CS 354 - Machine Organization & Programming
## Tuesday Sept 27 and Thursday Sept 29, 2022

**Midterm Exam - Thursday, October 6, 7:30 - 9:30 pm**

- **Room: Students will be assigned a room based on their Last "Family" name and Lec**
- **UW ID required**
- **#2 pencils required**
- **closed book, no notes, no electronic devices (e.g., calculators, phones, watches)**
- **see "Midterm Exam 1" on course site Assignments for topics**

**Project p2A:** Due on or before Friday, Sept 30th

**Project p2B:** Due on or before Friday, Oct 7

**Homework hw1:** Due on Monday Sept 26th (solution available Wed morning)

**Homework hw2:** Due on Monday Oct 3rd (solution available Wed morning)

**Last Week**

Command-line Arguments
Recall 2D Arrays
2D Arrays on the Heap
2D Arrays on the Stack
2D Arrays: Stack vs. Heap
Array Caveats

Meet Structures
Nesting in Structures and Arrays of Structures

**This Week**

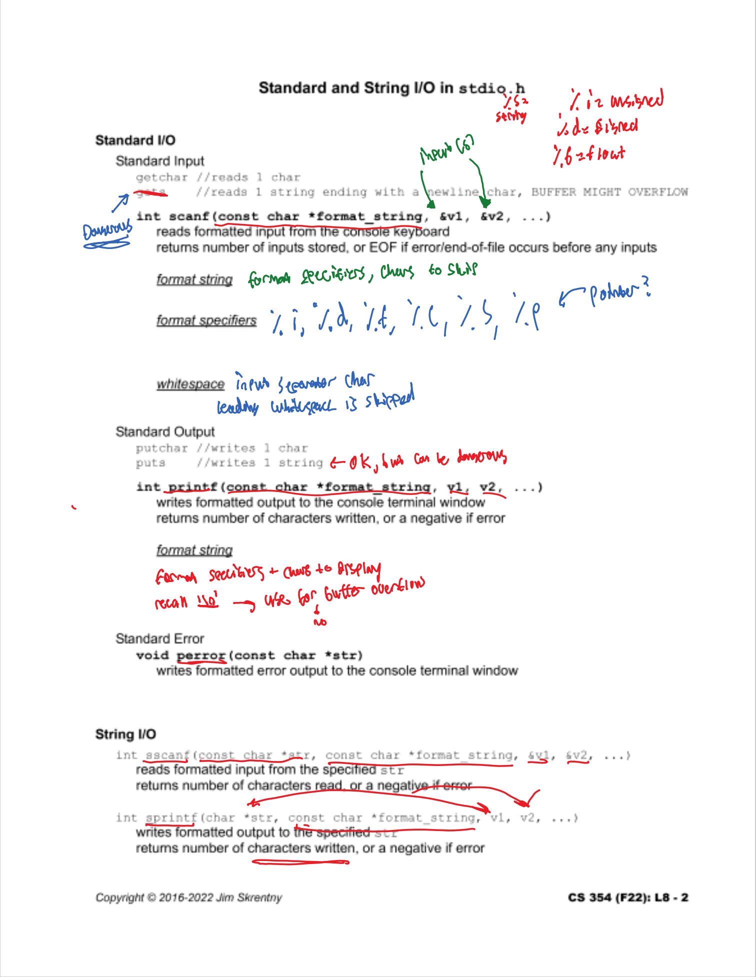
Passing Structures (from L6)
Pointers to Structures (from L6)
Standard & String I/O in `stdio.h`
File I/O in `stdio.h`
Copying Text Files

Three Faces of Memory

Virtual Address Space
C's Abstract Memory Model

Meet Globals and Static Locals
Where Do I Live?
Linux: Processes and Address Spaces
Exam Sample Cover Page

**Next Week**: The Heap & Dynamic Memory Allocators (p3)
Read: B&O 9.1, 9.2, 9.9.1-9.9.6
9.1 Physical and Virtual Addressing
9.2 Address Spaces
9.9 Dynamic Memory Allocation
9.9.1-9.9.6



## Standard and String I/O in `stdio.h`

%s = string

%i = unsigned
%d = signed
%f = float

**Standard I/O**

Standard Input

```
getchar //reads 1 char
~~gets~~    //reads 1 string ending with a newline char, BUFFER MIGHT OVERFLOW
```

Dangerous

Input(s)

```
int scanf(const char *format_string, &v1, &v2, ...)
```

reads formatted input from the console keyboard
returns number of inputs stored, or EOF if error/end-of-file occurs before any inputs

*format string* format specifiers, chars to skip

*format specifiers* %i, %d, %f, %c, %s, %p ← pointer?

*whitespace* input separator char
leading whitespace is skipped

Standard Output

```
putchar //writes 1 char
puts    //writes 1 string
```

← OK, but can be dangerous

```
int printf(const char *format_string, v1, v2, ...)
```

writes formatted output to the console terminal window
returns number of characters written, or a negative if error

*format string*

format specifiers + chars to display
recall '\0' → use for buffer overflow
no

Standard Error

```
void perror(const char *str)
```

writes formatted error output to the console terminal window

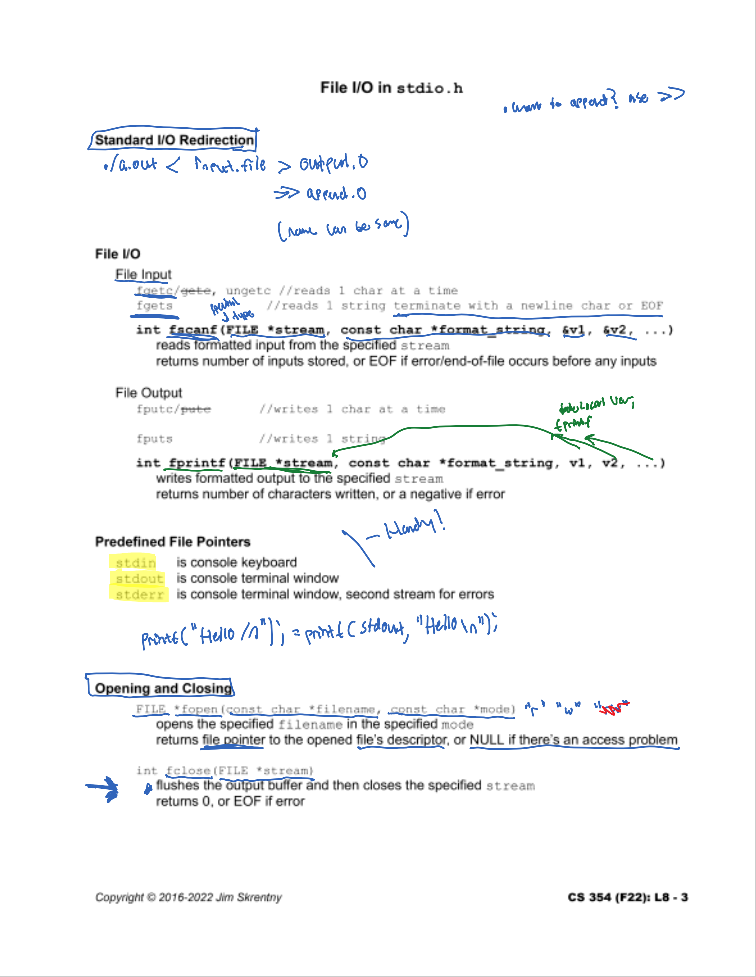
**String I/O**

```
int sscanf(const char *str, const char *format_string, &v1, &v2, ...)
```

reads formatted input from the specified `str`
returns number of characters read, or a negative if error

```
int sprintf(char *str, const char *format_string, v1, v2, ...)
```

writes formatted output to the specified `str`
returns number of characters written, or a negative if error



## File I/O in stdio.h

• want to append? use >>

### Standard I/O Redirection

./a.out < input.file > output.o

>> append.o

(name can be same)

### File I/O

File Input

fgetc/~~getc~~, ungetc //reads 1 char at a time

fgets special J dups //reads 1 string terminate with a newline char or EOF

**int fscanf(FILE *stream, const char *format_string, &v1, &v2, ...)**

reads formatted input from the specified stream

returns number of inputs stored, or EOF if error/end-of-file occurs before any inputs

File Output

fputc/~~putc~~ //writes 1 char at a time

data/Local Var, fprintf

fputs //writes 1 string

**int fprintf(FILE *stream, const char *format_string, v1, v2, ...)**

writes formatted output to the specified stream

returns number of characters written, or a negative if error

### Predefined File Pointers

— Handy!

stdin is console keyboard

stdout is console terminal window

stderr is console terminal window, second stream for errors

printf("Hello /n"); = printf(stdout, "Hello \n");

### Opening and Closing

FILE *fopen(const char *filename, const char *mode) "r" "w" "~~[illegible]~~"

opens the specified filename in the specified mode

returns file pointer to the opened file's descriptor, or NULL if there's an access problem

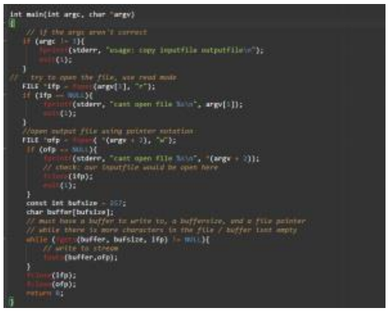
int fclose(FILE *stream)

→ flushes the output buffer and then closes the specified stream

returns 0, or EOF if error

## Copying Text Files

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

int main(int argc, char *argv[]) {

   if (argc != 3) {
      fprintf(stderr, "Usage: copy inputfile outputfile\n");
      exit(1);
   }


   FILE *ifp =
   if (ifp == NULL) {
      fprintf(stderr, "Can't open input file %s!\n", argv[1]);
      exit(1);
   }


   FILE *ofp =
   if (ofp == NULL) {
      fprintf(stderr, "Can't open output file %s!\n", argv[2]);


      exit(1);
   }


   const int bufsize = 257; //WARNING: assumes lines <= 256 chars
   char buffer[bufsize];










   return 0;
}
```



```
int main(int argc, char *argv)
{
    // if the argc aren't correct
    if (argc != 3){
        fprintf(stderr, "usage: copy inputfile outputfile\n");
        exit(1);
    }
//   try to open the file, use read mode
    FILE *ifp = fopen(argv[1], "r");
    if (ifp == NULL){
        fprintf(stderr, "cant open file %s\n", argv[1]);
        exit(1);
    }
    //open output file using pointer notation
    FILE *ofp = fopen( *(argv + 2), "w");
     if (ofp == NULL){
        fprintf(stderr, "cant open file %s\n", *(argv + 2));
        // check: our inputfile would be open here
        fclose(ifp);
        exit(1);
     }
     const int bufsize = 257;
     char buffer[bufsize];
     // must have a buffer to write to, a buffersize, and a file pointer
     // while there is more characters in the file / buffer isnt empty
     while ( fgets(buffer, bufsize, ifp) != NULL){
        // write to stream
        fputs(buffer,ofp);
     }
     fclose(ifp);
     fclose(ofp);
     return 0;
}
```

}

# Three Faces of Memory

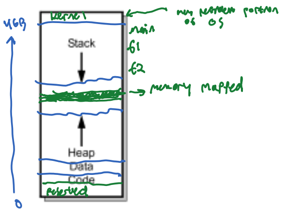
❋ *Abstraction:* Manage complexity by [illegible] on details

**Process View = Virtual Memory**

Goal: Provide simple view to programmers

*virtual address space (VAS)*: illusion provided by OS: each app has it's own memory space

*virtual address*: Simulated address that process generates



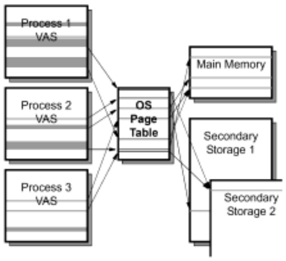
**System View = Illusionist (CS 537)**

Goal: make memory shareable and secure

*pages*: mem blocks managed by OS (4GB)

*page table*: OS data structure that maps virtual pages to physical pages of memory

- secure
- efficient



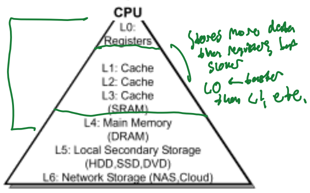
**Hardware View = Physical Memory**

Goal: Keep CPU Busy!

*physical address space (PAS)*: multi-level hierarchy, that ensures frequently accessed data is close to CPU

we cover this in 354

*physical address*: actual address used to access machine memory



CPU
L0: Registers
L1: Cache
L2: Cache
L3: Cache
(SRAM)
L4: Main Memory
(DRAM)
L5: Local Secondary Storage
(HDD,SSD,DVD)
L6: Network Storage (NAS,Cloud)
stores more data than registers but slower
L0 ← faster than L1, etc.

## Virtual Address Space (IA-32/Linux)

**32-bit Processor = 32-bit Addresses => $2^{32}$ = 4,294,967,296 = 4GB Address Space**

`11111111111111111111111111111111 = 0xFFFFFFFF`

*address space*: range of valid space for process

*process*: a running program

`11000000000000000000000000000000 = 0xC0000000`

C 0 0 0 0 0 0 0

*kernel*: memory part of operating system

*user process*: any process not related to kernel

- *Every user process* has a simple memory view: this makes coding easier.

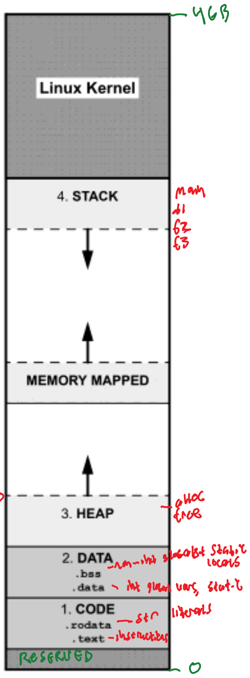
for DLL files, file I/O → large heap alloc.

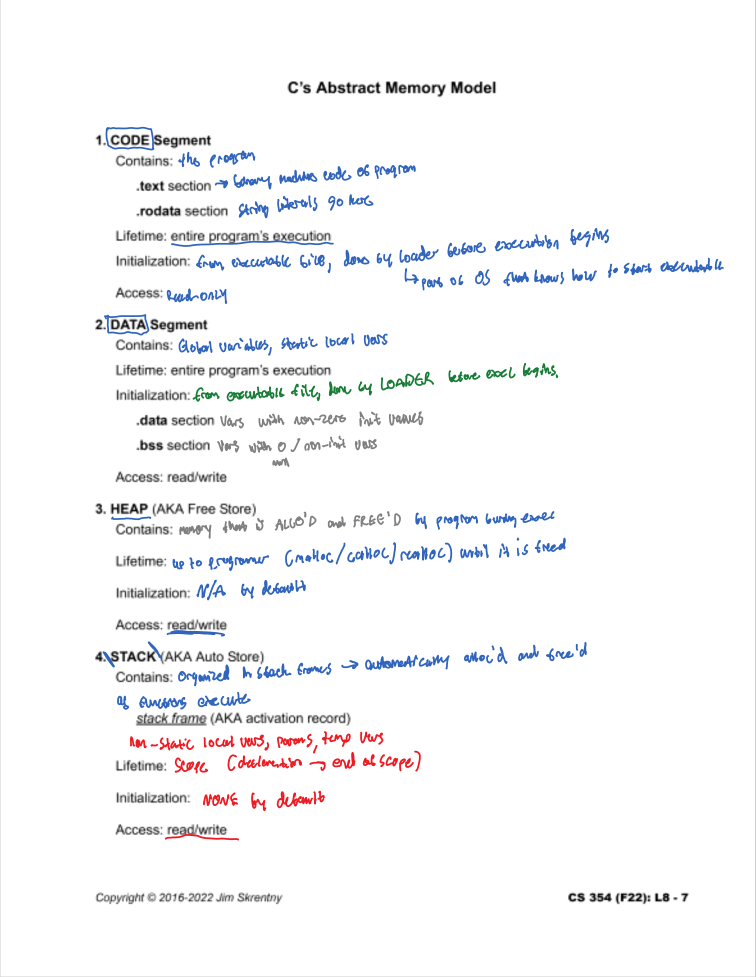
brk → keeps track of what's on heap

starts here!

`00001000000001001000000000000000 = 0x08048000`

`00000000000000000000000000000000 = 0x00000000`





## C's Abstract Memory Model

1. **CODE Segment**

   Contains: the program

   **.text** section → binary machine code of program

   **.rodata** section string literals go here

   Lifetime: entire program's execution

   Initialization: from executable file, done by loader before execution begins
   ↳ part of OS that knows how to start executable

   Access: Read-ONLY

2. **DATA Segment**

   Contains: Global variables, static local vars

   Lifetime: entire program's execution

   Initialization: from executable file, done by LOADER before exec begins.

   **.data** section vars with non-zero init values

   **.bss** section vars with 0 / non-init vars
   init

   Access: read/write

3. **HEAP** (AKA Free Store)

   Contains: memory that is ALLOC'D and FREE'D by program during exec

   Lifetime: up to programmer (malloc/calloc/realloc) until it is freed

   Initialization: N/A by default

   Access: read/write

4. **STACK** (AKA Auto Store)

   Contains: Organized in stack frames → automatically alloc'd and free'd as functions execute

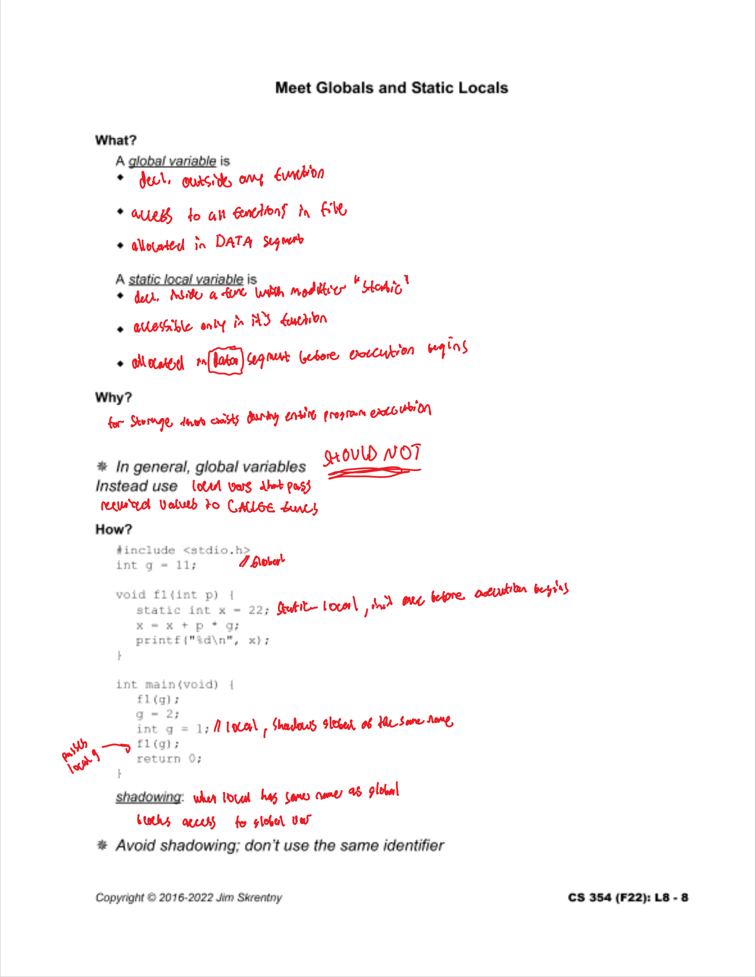
   *stack frame* (AKA activation record)

   non-static local vars, params, temp vars

   Lifetime: Scope (declaration → end of scope)

   Initialization: NONE by default

   Access: read/write



## Meet Globals and Static Locals

**What?**

A *global variable* is

- decl. outside any function
- access to all functions in file
- allocated in DATA segment

A *static local variable* is

- decl. inside a func with modifier "static"
- accessible only in it's function
- allocated in $\boxed{\text{data}}$ segment before execution begins

**Why?**

for storage that exists during entire program execution

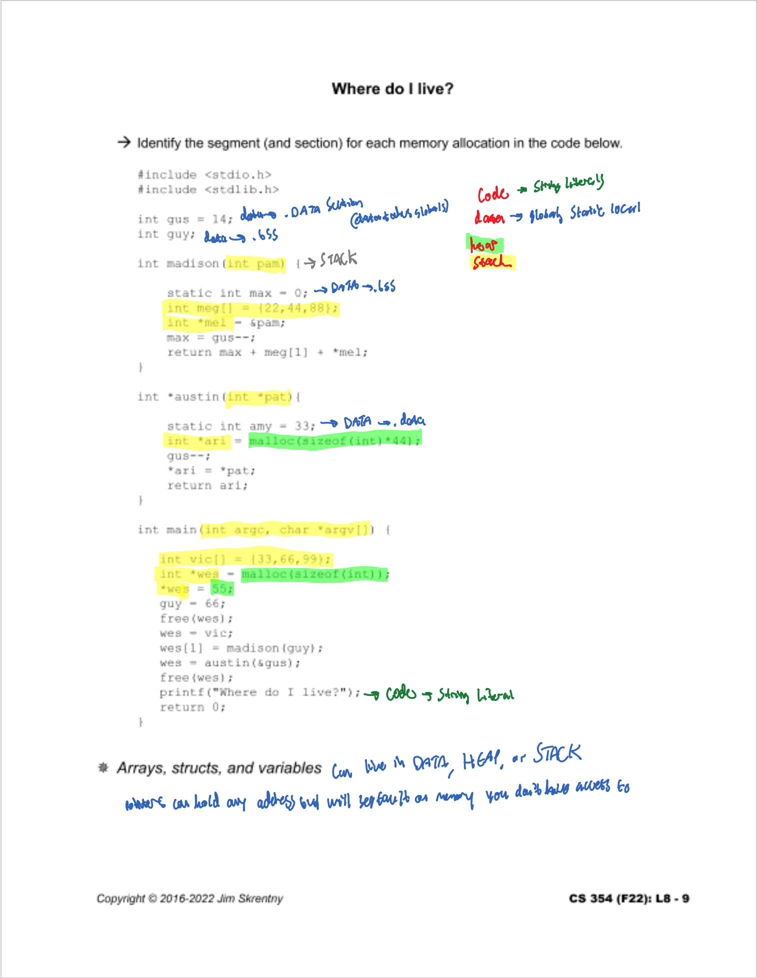
✻ *In general, global variables* SHOULD NOT
*Instead use* local vars that pass required values to CALLEE funcs

**How?**

```
#include <stdio.h>
int g = 11;        // Global

void f1(int p) {
   static int x = 22; static local, init once before execution begins
   x = x + p * g;
   printf("%d\n", x);
}

int main(void) {
   f1(g);
   g = 2;
   int g = 1; // local, shadows global of the same name
passes local g → f1(g);
   return 0;
}
```

*shadowing*: when local has same name as global
blocks access to global var

✻ *Avoid shadowing; don't use the same identifier*



## Where do I live?

→ Identify the segment (and section) for each memory allocation in the code below.

Code → String literals

data → global, Static local

heap

stack

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

int gus = 14;  data → .DATA Section
                    (data initialized globals)
int guy;  data → .bss

int madison(int pam)  { → STACK

    static int max = 0;  → DATA → .bss
    int meg[] = {22,44,88};
    int *mel = &pam;
    max = gus--;
    return max + meg[1] + *mel;
}

int *austin(int *pat){

    static int amy = 33;  → DATA → .data
    int *ari = malloc(sizeof(int)*44);
    gus--;
    *ari = *pat;
    return ari;
}

int main(int argc, char *argv[]) {

   int vic[] = {33,66,99};
   int *wes = malloc(sizeof(int));
   *wes = 55;
   guy = 66;
   free(wes);
   wes = vic;
   wes[1] = madison(guy);
   wes = austin(&gus);
   free(wes);
   printf("Where do I live?");  → code → String literal
   return 0;
}
```

✸ *Arrays, structs, and variables* can live in DATA, HEAP, or STACK

pointers can hold any address but will segfault on memory you don't have access to

# Linux: Processes and Address Spaces

## Process and Job Control

- Linux is a multi-tasking OS where you can run things concurrently

`ps` lists snapshot of user processes

`jobs` lists only processes started from cmd line

`&` puts process in background
ctrl+z suspend prog

`bg` put suspended prog in background
`fg` put in foreground

ctrl+c break

top  disp table of process resource usage

## Program Size

size <executable or object_file> display size of prog's mem segments + total size
a.out → compiled size (not running)

```
$gcc -m32 myProg.c
$size a.out
   text    data     bss     dec     hex filename
   1029     276       4    1309     51d a.out
```

TOTAL

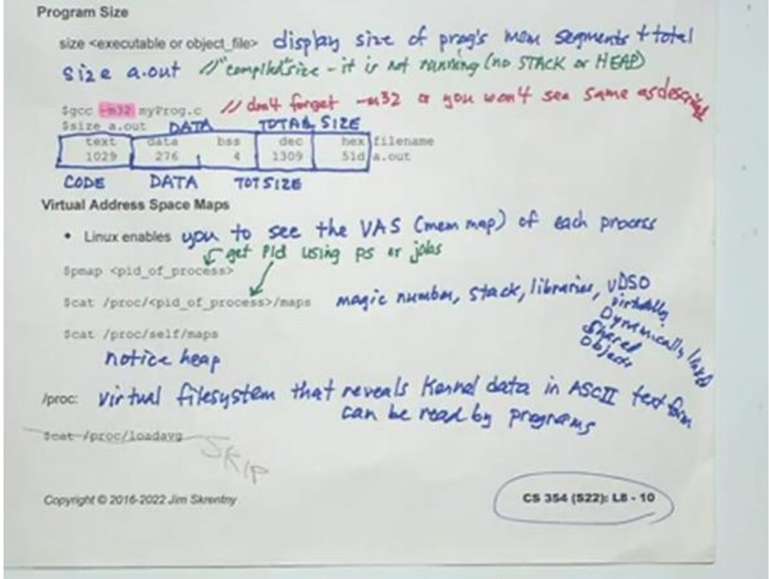
code  data

## Virtual Address Space Maps

- Linux enables you to see VAS (mem map) of each process

```
$pmap <pid_of_process>

$cat /proc/<pid_of_process>/maps

$cat /proc/self/maps
```

/proc:



## Program Size

size <executable or object_file> display size of prog's mem segments + total
size a.out // "compiled" size - it is not running (no STACK or HEAP)

```
$gcc -m32 myProg.c
$size a.out
   text    data     bss     dec     hex filename
   1029     276       4    1309     51d a.out
```

// don't forget -m32 or you won't see same as describ

DATA  TOTAL SIZE

CODE  DATA  TOT SIZE

## Virtual Address Space Maps

- Linux enables you to see the VAS (mem map) of each process

get pid using ps or jobs

```
$pmap <pid_of_process>

$cat /proc/<pid_of_process>/maps

$cat /proc/self/maps
```

magic number, stack, libraries, VDSO
virtually Dynamically linked Shared Objects

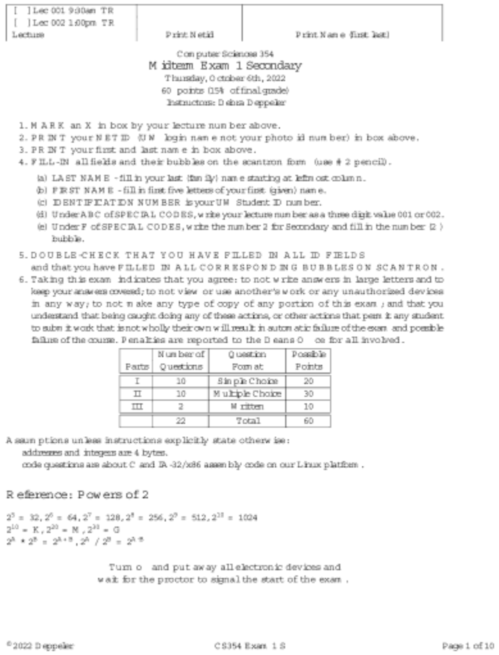
notice heap

/proc: virtual filesystem that reveals Kernel data in ASCII text form
can be read by programs

~~$cat /proc/loadavg~~ Skip

Copyright © 2016-2022 Jim Skrentny

CS 354 (S22): L8 - 10



| | | |
|---|---|---|
| [ ] Lec 001 9:30am TR<br>[ ] Lec 002 1:00pm TR<br>Lecture | Print Netid | Print Name (first last) |

Computer Sciences 354

## Midterm Exam 1 Secondary

Thursday, October 6th, 2022

60 points (15% of final grade)

Instructors: Debra Deppeler

1. MARK an X in box by your lecture number above.
2. PRINT your NETID (UW login name not your photo id number) in box above.
3. PRINT your first and last name in box above.
4. FILL-IN all fields and their bubbles on the scantron form (use # 2 pencil).
   - (a) LAST NAME - fill in your last (family) name starting at leftmost column.
   - (b) FIRST NAME - fill in first five letters of your first (given) name.
   - (c) IDENTIFICATION NUMBER is your UW Student ID number.
   - (d) Under ABC of SPECIAL CODES, write your lecture number as a three digit value 001 or 002.
   - (e) Under F of SPECIAL CODES, write the number 2 for Secondary and fill in the number (2) bubble.
5. DOUBLE-CHECK THAT YOU HAVE FILLED IN ALL ID FIELDS and that you have FILLED IN ALL CORRESPONDING BUBBLES ON SCANTRON.
6. Taking this exam indicates that you agree: to not write answers in large letters and to keep your answers covered; to not view or use another's work or any unauthorized devices in any way; to not make any type of copy of any portion of this exam; and that you understand that being caught doing any of these actions, or other actions that permit any student to submit work that is not wholly their own will result in automatic failure of the exam and possible failure of the course. Penalties are reported to the Deans Office for all involved.

| Parts | Number of Questions | Question Format | Possible Points |
|---|---|---|---|
| I | 10 | Simple Choice | 20 |
| II | 10 | Multiple Choice | 30 |
| III | 2 | Written | 10 |
| | 22 | Total | 60 |

Assumptions unless instructions explicitly state otherwise:

addresses and integers are 4 bytes.

code questions are about C and IA-32/x86 assembly code on our Linux platform.

### Reference: Powers of 2

$2^5 = 32, 2^6 = 64, 2^7 = 128, 2^8 = 256, 2^9 = 512, 2^{10} = 1024$

$2^{10} = K, 2^{20} = M, 2^{30} = G$

$2^A * 2^B = 2^{A+B}, 2^A / 2^B = 2^{A-B}$

Turn off and put away all electronic devices and
wait for the proctor to signal the start of the exam.

© 2022 Deppeler CS354 Exam 1 S Page 1 of 10