CS 354 - Machine Organization & Programming
Tuesday Jan 30th and Thursday Feb 1st, 2024

Project p1: DUE on or before Tuesday 2/6, available until Sunday 2/11

- See **PM Activities** for days and times for **BYOL: Linux Basics** this week.

Project p2A: Released this week Friday

Homework hw1: Assigned soon

Exam Conflicts: Report for e1,e2,e3 by 2/9 : **http://tiny.cc/cs354-conflicts**

TA Lab Consulting Available. See link on course front page.

Week 2 Learning Objectives (at a minimum be able to)

- state and show in memory diagrams the name, value, type, address, size of variable
- understand and show binary representation and byte ordering for int, char, address, values
- declare, assign, and dereference pointer variables
- use **stdlib.h** functions *malloc* and *free* to manage dynamically allocated "heap" memory
- code, describe, and diagram 1D arrays on stack and on heap
- understand and show byte representation of character arrays and C strings
- understand and use **string.h** library functions with string literals and C strings

This Week

`Tuesday`	`Thursday`
Finish COMPILE, RUN, DEBUG Recall Variables and Meet Pointers Practice Pointers Recall 1D Arrays 1D Arrays and Pointers	Passing Addresses 1D Arrays on the Heap Pointer Caveats Meet C Strings Meet `string.h`
`Read before Thursday` K&R Ch. 7.8.5: Storage Management (malloc and calloc) K&R Ch. 5.5: Character Pointers and Functions K&R Ch. 5.6: Pointer Arrays; Pointers to Pointers	

Next Week

Topic: 2D Arrays and Pointers

Read:

K&R Ch. 5.7: Multi-dimensional Arrays
K&R Ch. 5.8: Initialization of Pointer Arrays
K&R Ch. 5.9: Pointers vs. Multi-dimensional Arrays
K&R Ch. 5.10: Command-line Arguments

Do: Finish project p1 and start p2A

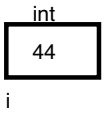
Recall Variables

What? A scalar variable is primitive a unit of storage whose contents can change.

→ Draw a basic memory diagram for the variable in the following code:

```
void someFunction(){
   int i = 44;
```

basic memory diagram:



Aspects of a Variable

identifier: name of variable

stores 44 in 2's complement

value: data stored in variable

type: how its gonna be represented, interpreted

address: starting location in memory

integer, 32 bits, 4 bytes

size: number of bytes, for integers usually 4

❋ *A scalar variable used as a source operand* thats where data alr exists, gonna read that value from storage

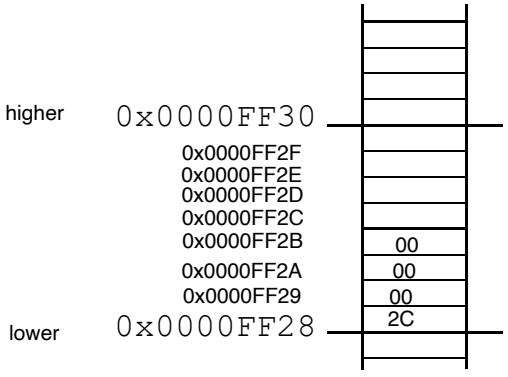
e.g., `printf("%i\n", i);` read i and output

❋ *A scalar variable used as a destination operand* going to store

e.g., `i = 11;` writes value to storage

Linear Memory Diagram

A linear memory diagram is



if 44 stored at 28

most significant least

base 10							**44**	decimal
base 2	0000	0000	0000	0000	0000	0000	0010	1100 binary
base 16	0	0	0	0	0	0	2	C hex

least significant byte
8 bits

byte addressability: each address identifies one byte

only talk about starting address for 4 bytes, for integer == 4 bytes, so address is FF28

endianess: byte ordering for variables with more than 1 byte, eg int

cs354 ***little endian:*** least significant byte in the lowest address

big endian: opposite — most significant byte in the lowest address

Meet Pointers

What? A *pointer* variable is

- a scalar variable whose value is a memory address
- similar to Java reference

Why?

- for indirect access to memory
- for indirect access to functions
- common in C libraries
- access to memory mapped hardware

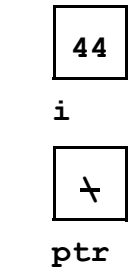
How?

→ Consider the following code:

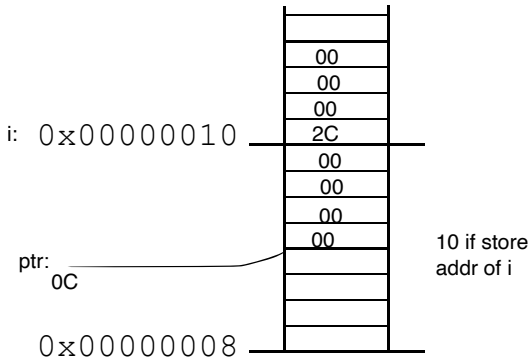
```
void someFunction(){
   int i = 44;

   int *ptr = NULL;
```

if want to point ptr to address of i
ptr = &i;

Basic Diag.



Linear Diag.



→ What is `ptr`'s initial value? address? type? size?

NULL or 0x0 | 0x0000000C | int * | 4 bytes

pointer: contains the address, and does the pointing

pointee: whats pointed at

`&` *address of* operator: returns the address of its operand &a returns address of variable a

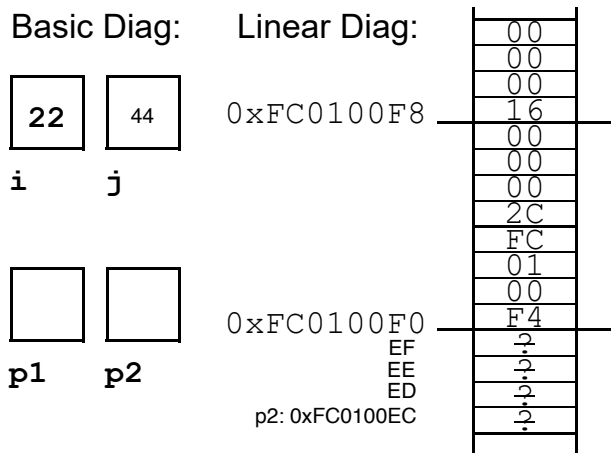
`*` *dereferencing* operator: follow address to pointee, access pointee via the pointer

Practice Pointers

22 :0000 0000 0001 0110 or 00 00 00 16
44 :0000 0000 0010 1100 or 00 00 00 2C

→ Complete the following diagrams and code so that they all correspond to each other:

```
void someFunction(){
   int i = 22
   int j = 44;
   int *p1 = &j;
   int *p2;  //at addr 0xFC0100EC
```



→ What is `p1`'s value?

0xFC0100F4

→ Write the code to display `p1`'s pointee's value.

printf("%p\n", *p1);

→ Write the code to display `p1`'s value.

printf("%p\n", p1);

→ Is it useful to know a pointer's exact value?

→ What is `p2`'s value?

→ Write the code to initialize `p2` so that it points to nothing.

int *p2 = NULL;

→ What happens if the code below executes when `p2` is NULL?

```
printf("%i\n", *p2);
```

→ What happens if the code below executes when `p2` is uninitialized?

```
printf("%i\n", *p2);
```

→ Write the code to make `p2` point to `i`.

→ How many pointer variables are declared in the code below?

```
void someFunction(){
   int* p1, p2;
```

→ What does the code below do?

```
int **q = &p1;
```

Recall 1D Arrays

What? An *array* is

- a compound unit of storage with elements of some type
- access via identifier and indexing
- allocated as contiguous fixed size block of memory

Why?

- store a collection of data of same type with fast access
- easier to declare than individual variables for each item

How?

```
void someFunction(){
   int a[5];
```

→ How many integer elements have been allocated memory? 5

heap changes while prog is running

→ Where in memory was the array allocation made? not all arrays are on heap

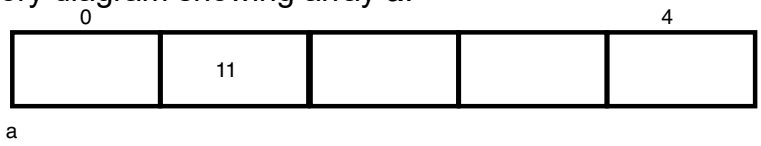
STACK ALLOC ARRAY

programmer manages heap, compilers managed stack

→ Write the code that gives the element at index 1 a value of 11.

a[1] = 11;

→ Draw a basic memory diagram showing array `a`.



❋ *In C, the identifier for a stack allocated array (SAA)* IS NOT A VARIABLE

❋ *A SAA identifier used as a source operand* provides the array address

e.g., `printf("%p\n", a);`

❋ *A SAA identifier used as a destination operand*

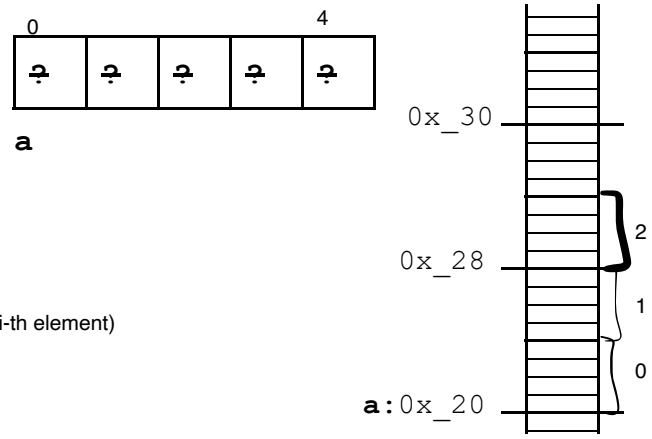
~~a = new int[3];~~

a = ???
COMPILER ERROR, cant change a,
can change element using index of a

1D Arrays and Pointers

Given:

```
void someFunction(){
   int a[5];   // SAA
```



Address Arithmetic IS FAST!

❋ *a[i]* equivalent *(a + i) (a + i is take a and go to i-th element)

1. compute the address

   start at a's beginning address: 0x_20

   add byte offset to get to element i: scaled by element size in bytes (in this case 4 bytes since int, if char then 1, if double then 8)

   compiler does scaling for us, no need specify

2. dereference the computed address to access the element

   add * in front a + i

   *(a + i)

→ Write address arithmetic code to give the element at index 3 a value of 33.

*(a + 3) = 33;

→ Write address arithmetic code equivalent to `a[0] = 77;`

*(a + 0) = 77 or *a = 77

Using a Pointer

77 is 0x4D

→ Write the code to create a pointer `p` having the address of array `a` above.

int *p = a; (since a is already an address)

→ Write the code that uses p to give the element in `a` at index 4 a value of 44.

*(p + 4) = 44;

❋ *In C, pointers and arrays* are closely related but not the same

Passing Addresses

Recall ***Call Stack Tracing***:

- manually trace function calls

- each function gets a box (stack frame)

- "top" box is currently running

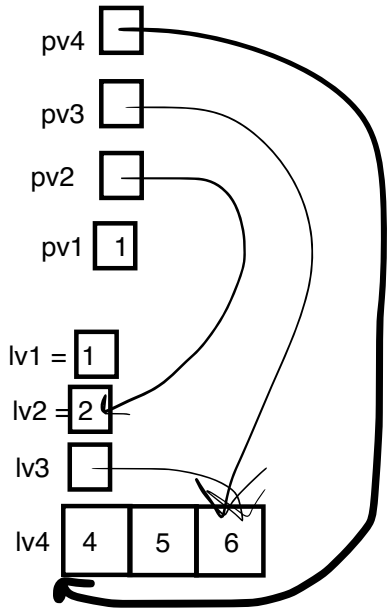
➢ What is output by the code below?

```
void f(int pv1, int *pv2, int *pv3, int pv4[]) {
   int lv = pv1 + *pv2 + *pv3 + pv4[0];
   pv1    = 11;
   *pv2   = 22;
   *pv3   = 33;
   pv4[0] = lv;
   pv4[1] = 44;
}
int main(void) {
   int lv1 = 1, lv2 = 2;
   int *lv3;
   int lv4[] = {4,5,6};
   lv3 = lv4 + 2;
   f(lv1, &lv2, lv3, lv4);
   printf("%i,%i,%i\n",lv1,lv2,*lv3);
   printf("%i,%i,%i\n",lv4[0],lv4[1],lv4[2]);
   return 0;
}
```

dont know what is in lv3
just a pointer to an int type

lv3 will now point to 6 since lv4 points to 4

pass copies of values to function f



Pass-by-Value

- scalars: param is a scalar variable that gets a copy of its scalar argument lv1 is an example
- pointers: param is a pointer variable that gets copy of address arguemtn (lv2 is an example)
- arrays: param is a pointer variable that gets copy of array address argument

❋ *Changing a callee's parameter* changes the callee's copy only (eg if change pv3 to point to sth else, does not change 6 in lv4)

❋ *Passing an address* requires caller trusts callee, since callee can modify data pointed to

1D Arrays on the Heap

What? Two key memory segments used by a program are the

STACK
static (fixed in size) allocations
allocation size known during compile time

and HEAP
dynamic allocated memory
happens at "runtime", need size

Why? Heap memory enables

- access to more memory than available at compile time
- us to have blocks of memory that can be allocated and freed

How? #include <stdlib.h>

`void* malloc(size_in_bytes)` function that reserves a block of heap memory of a specified size
returns (void*) a generic pointer, since we did not tell it what type it is

why void pointer work almost exactly to char pointer but not int pointer,
both void and char point to 1 byte, address arithmetic, int pointer cast by 4, both void and char cast by 1

`void free(void* ptr)` frees heap block that pointer points to
heap allocater knows how big the heap block is

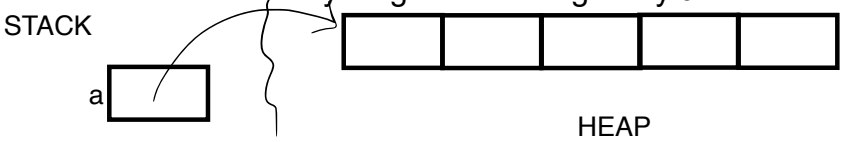
`sizeof(operand)` returns number of bytes of operand

→ For IA-32 (x86), what value is returned by `sizeof(double)`? `sizeof(char)`? `sizeof(int)`?
8 bytes 1 bytes 4 bytes

→ Write the code to dynamically allocate an integer array named `a` having 5 elements.

```
void someFunction(){
    int *a = malloc(sizeof(int) * 5);
}
```

→ Draw a memory diagram showing array `a`.



→ Write the code that gives the element at indexes 0, 1 and 2 a values of 0, 11 and 22
by using pointer dereferencing, indexing, and address arithmetic respectively.

ptr dereferencing: `*a = 0;`

indexing: `a[1] = 11;`

addr arithmetic: `*(a+2) = 22;`

→ Write the code that uses a pointer named `p` to give the element at index 3 a value of 33.

```
int *p;
p = a;
p[3] = 33; or *(p+3) = 33;
```

→ Write the code that frees array `a`'s heap memory.

free(a); or free(p); works too

if a[4] = 4; is executed, this is called dangling pointer, pointer to memory that has been freed. in theory should crash, but in practice might store something else there

Pointer Caveats

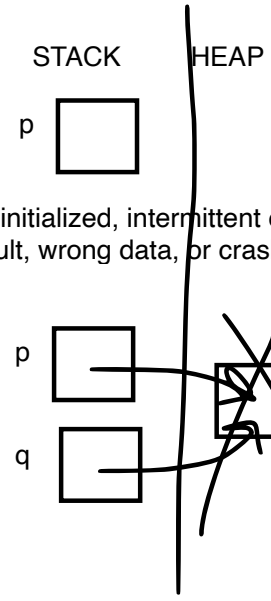
❋ *Don't dereference uninitialized or NULL pointers!*

```
int *p;
*p = 11;
```
uninitialized

```
int *q = NULL;
*q = 11;
```
NULL

ALWAYS SEG FAULT

NULL is like 0, cannot dereference 0, no perms so seg fault



❋ *Don't dereference freed pointers!*

```
int *p = malloc(sizeof(int));
int *q = p;
. . .
free(p);
. . .
*q = 11;
```
intermittent error, dangling

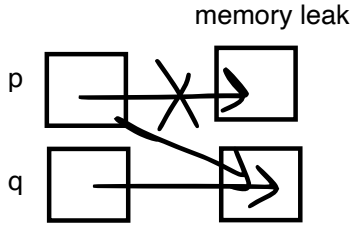
p is uninitialized, intermittent error seg fault, wrong data, or crash

dangling pointer: pointer variable that contains address to heap memory that has been freed

❋ *Watch out for heap memory leaks!*

memory leak: memory that is unusable since it was not freed correctly

```
int *p = malloc(sizeof(int));
int *q = malloc(sizeof(int));
. . .
p = q;
```



❋ *Be careful with testing for equality!*

assume p and q are pointers

p = q	compares nothing because it's assignment
p == q	compares values in pointers
*p == *q	compares values in pointees

*p == q results in type error, cannot compare int and pointer

❋ *Don't return addresses of local variables!*

local variable goes away when function is done

```
int *ex1() {
   int i = 11;  // local on stack
   return &i;   // mem not avail after function ends
}

int *ex2(int size) {
   int a[size];   // SAA
   return a;      // cannot return address from the stack
}
```

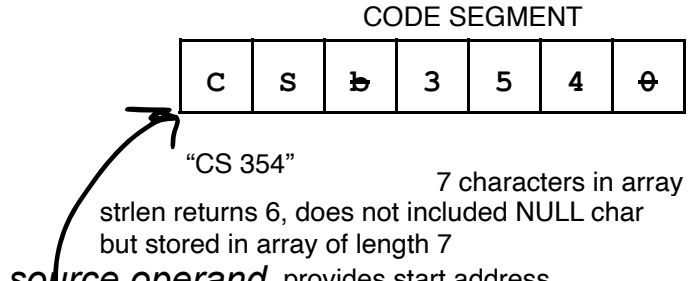
Meet C Strings

What? A ***string*** is

- a sequence of characters terminated with '\0' a NULL character
- a 1D array of characters of char with string length (strlen) = strlen + 1

What? A ***string literal*** is

- a constant source code string
- allocated prior to execution

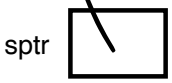


❋ *In most cases, a string literal used as a source operand* provides start address

How? Initialization

```
void someFunction(){
   char *sptr = "CS 354";
```

STACK



→ Draw the memory diagram for `sptr`.

→ Draw the memory diagram for `str` below.

```
   char str[9] = "CS 354";
```

str is addr of first letter of array, how many letters? 9

copies "CS 354" into array str,
all initialized to '\0' originally

STACK

→ During execution, where is `str` allocated?

STACK

How? Assignment

→ Given `str` and `sptr` declared in `somefunction` above,
what happens with the following code?

no type in this cases

```
sptr = "mumpsimus";
```

can i assign to charaacter pointer new string literal? yes ok
string literal "mumpsimus" exist in same code segment as above

```
str = "folderol";
```

not ok
assignment of string literal copies each letter into array, but not in re-assignment
can copy each letter by letter in, using loop (?)

compiler error — cannot assign new string literal

❋ *Caveat: Assignment cannot be used* to copy new character arrays

Meet `string.h`

What? *`string.h`* is collection of useful functions in manipulating C strings

`int strlen(const char *str)`
Returns the length of string `str` up to but *not* including the null character.

`int strcmp(const char *str1, const char *str2)`
Compares the string pointed to by `str1` to the string pointed to by `str2`.
returns: < 0 (a negative) if str1 comes before str2
0 if str1 is the same as str2
>0 (a positive) if str1 comes after str2

`char *strcpy(char *dest, const char *src)`
Copies the string pointed to by `src` to the memory pointed to by `dest`
and terminates with the null character.

`char *strcat(char *dest, const char *src)`
Appends the string pointed to by `src` to the end of the string pointed to by `dest`
and terminates with the null character.

❋ *Ensure the destination character array* is large enough for result and '\0'

buffer overflow: exceed the bounds of the array
not compiler error, not even runtime error, might just be bad data

How? `strcpy`

→ Given `str` and `sptr` as declared in `somefunction` on the previous page, what happens with the following code?

```
strcpy(str, "folderol");
```
yes can copy all into string, does not exceed bounds, fits 9 characters (including null char) perfectly

```
strcpy(str, "formication");
```
buffer overflow — only fits "formicat"

```
strcpy(sptr, "vomitory");
```
doesnt matter if fit, cannot copy into code segment (READ ONLY) SEG FAULT — will attempt to copy first char "v" into segment no permissions to wrtie to code segment

SEG FAULT is a runtime error, crashes program

❋ *Rather than assignment, strcpy (or strncpy) must be used to* copy C strings from one array to another
strncpy — define n characters to be copied

❋ *Caveat: Beware of* buffer overflow, and attempting to write to code segment