CS 354 - Machine Organization & Programming
Tuesday Jan 31 and Thursday Feb 2nd, 2023

Project p1: DUE on or before Friday 2/11 (get it done and submit this week if possible)
Project p2A: Released Friday and due on or before Friday 2/18 p2B will overlap
Homework hw1: Assigned soon
Exam Conflicts (check entire semester): Report by 2/11 to: **http://tiny.cc/cs354-conflicts**
TA Lab Consulting & PM Activities are scheduled. See links on course front page.

Last Week

Welcome Course Infor Getting Started in Linux EDIT COMPILE, RUN, DEBUG(see recordings)	C Program Structure (L2-6) C Logical Control Flow Recall Variables Meet Pointers Practice Pointers

This Week

`Tuesday`	`Thursday`
Practice Pointers (from L02) 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 (handin this week Friday to ensure time on p2A next week)
Start project p2A

Recall 1D Arrays

What? An *array* is

- a compound unit of storage having parts called elements
- accessed using identifiers and index (sometimes called offset)
- allocated as a contiguous fixed size block of memory

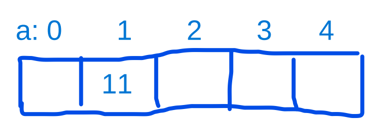
Why?

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

How?

a is NOT a pointer, stores address of element 0 of the array, but can't be changed

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



→ How many integer elements have been allocated memory? 5

→ Where in memory was the array allocation made? STACK (different from java)

→ 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*

e.g., `printf("%p\n", a);` // 0x _ _

❈ *A SAA identifier used as a destination operand* results in an error!

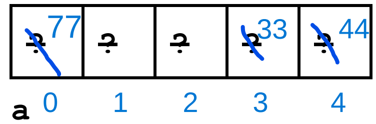
~~a~~= cannot change value of a (causes compiler error)

1D Arrays and Pointers

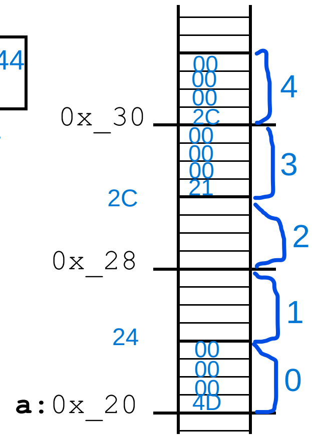
Given:

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

BASIC STACK



STACK



Address Arithmetic is FAST!

❋ *a[i]* ≡ *(a+i) scaled index

i*4 for int
i*1 for char
i*8 for double

1. compute the address

   start at beginning address
   add byte offset to get element i
   offset is scaled by size of (element type)

2. dereference the computed address to access the element

   ~~*a+i~~ *(a+i)

   Note: parentheses are required
   because * has higher precedence than addition

→ 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

~~a = 77~~

*(a) = 77
*a = 77

Using a Pointer

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

int *p = a; //&a
int *p; p = a;

→ 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

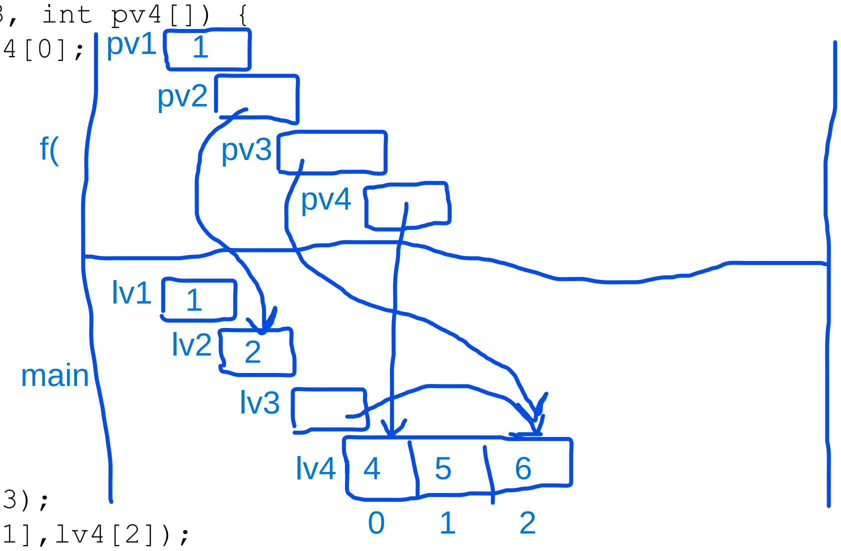
Passing Addresses

Recall *Call Stack Tracing*:

- manually trace code with functions
  in a manner that mimics the machine
- each function gets a box, called a "stack frame"
  which stores: parameters, local variables, temp variables, ... and more
- "top" box is running function
  those below are waiting for callee to return

➢ 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;
}
```



output
1, 22, 33
13, 44, 32

Pass-by-Value

- scalars: param is a scalar variable that gets a copy of its scalar argument
- pointers: param is a pointer variable that gets copy of address argument
- arrays: param is a pointer variable that gets a copy of any address (as long as it's the right type)

❈ *Changing a callee's parameter* changes callee's copy, not caller's value

❈ *Passing an address* requires TRUST, because callee can change caller value

1D Arrays on the Heap (like java)

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

STACK	and	HEAP
static (fixed in size) allocations		dynamic allocation during runtime
allocation size known during compile time		

Why? Heap memory enables

- access to more than available at compile time
- having blocks of memory to be allocated and freed

How? #include <stdlib.h>

`void* malloc(size_in_bytes)` memory allocator

function reserves a block of heap memory of specified size
returns a generic pointer

`void free(void* ptr)`

frees the heap block that pointer points to

`sizeof(operand)` returns the size in bytes of the operand

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

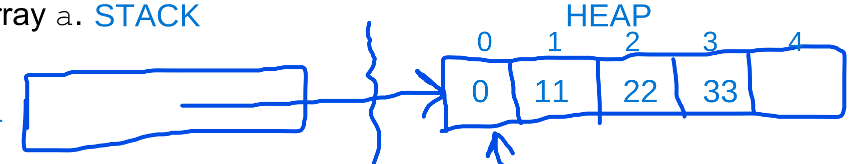
→ Write the code to dynamically allocate an integer array named `a` having 5 elements.

```
void someFunction(){
```

int * a = malloc(sizeof(int) * 5);

→ Draw a memory diagram showing array `a`. STACK HEAP

int * 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.

dereferencing: *a = 0;

indexing: a[1] = 11;

address arithmetic: *(a+2) = 22;

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

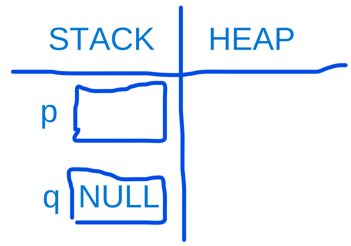
int *p = a;
*(p+3) = 33;

p

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

free(a);

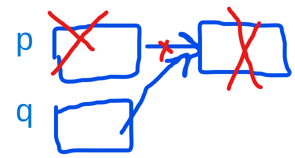
Pointer Caveats



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

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

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



❋ *Don't dereference freed pointers!*

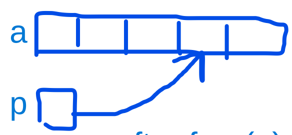
```
int *p = malloc(sizeof(int));
int *q = p;
. . .
free(p);
. . .
*q = 11;//SEG FAULT
        //or intermittent error
```

dangling pointer: A pointer ith address to heap memory that has been freed
in the above example, q is a dangling pointer

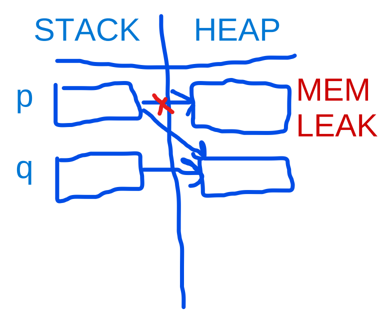
❋ *Watch out for heap memory leaks!*

memory leak: heap memory that is unusable since not freed properly

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



after free(a);
p is dangling ptr



❋ *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

❋ *Don't return addresses of local variables!* Why not? because they are local to function and are freed when function ends

```
int *ex1() {
   int i = 11;//local variable
   return &i; //memory not available after function ends
}

int *ex2(int size) {
   int a[size];//stack allocated array, local to function
   return a;    //not available after function ends
}
```

Meet "C Strings"

What? A *string* "C String" is

- a sequence of characters with a null terminating character '\0'
- allocated as 1D array of characters with min size = str length +1

What? A *string literal* is

C	S	~~b~~	3	5	4	~~0~~

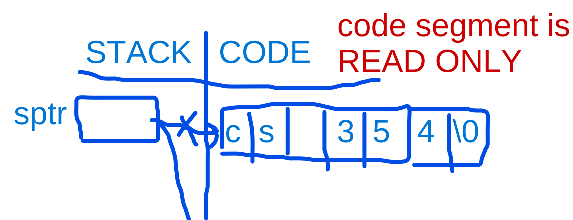
- a constant source code String ex. "CS 354"
- allocated prior to execution in the CODE segment (not stack or heap)

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

How? Initialization

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

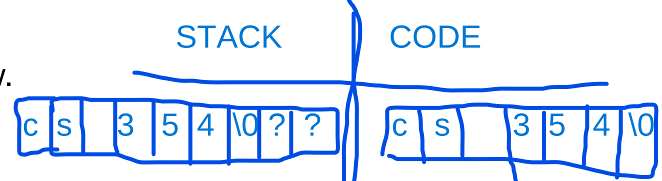
string literal



→ Draw the memory diagram for `sptr`.

→ Draw the memory diagram for `str` below.

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

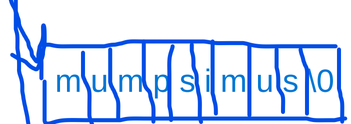


→ During execution, where is `str` allocated?
STACK

How? Assignment

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

`sptr = "mumpsimus";` OK



`str = "folderol";` //COMPILER ERROR
//cannot assign to stack allocated array

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

Meet `string.h`

What? *`string.h`* is a collection of useful functions to manipulate 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)` like java compareto()
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 the result and '\0' otherwise causes buffer overflow

buffer overflow: exceeding bounds of the array

How? `strcpy`

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

`strcpy(str, "folderol");` NO PROBLEM

`strcpy(str, "formication");` BUFFER OVERFLOW

`strcpy(sptr, "vomitory");` SEG FAULT - CODE segment is read-only

❋ *Rather than assignment, strcpy (or strncpy) must be used to*
copy from one char array to another

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