CS 354 - Machine Organization & Programming
Tuesday Sept 12th and Thursday Sept 14, 2023

Project p1: DUE on or before Friday 9/22 (submit this week if possible)

Project p2A: Released Friday and due on or before Friday 9/29

Homework hw1: Assigned soon

Exam Conflicts (check entire semester): Report by 9/29 to: **http://tiny.cc/cs354-conflicts**

TA Lab Consulting & PM Activities are scheduled. See links 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 "address" variables
- code, describe, and diagram 1D arrays on stack and on heap
- understand and show byte representation of character array vs "C string" variables
- understand and use <string.h> library functions with string literals and "C string" variables

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;
```

int i | 44 |

Aspects of a Variable

identifier: name

value: data stored.

type: representation of bit pattern.

address: starting location

size: num of bytes

❋ *A scalar variable used as a source operand* read value

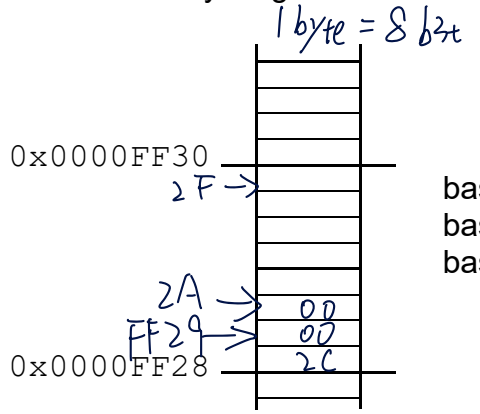
e.g., `printf("%i\n", i);`

❋ *A scalar variable used as a destination operand*

e.g., `i = 11;`

Linear Memory Diagram

A linear memory diagram is



	most significant							least
base 10								**44**
base 2	0000	0000	0000	0000	0000	0000	0010	1100
base 16	0	0	0	0	0	0	2	C

lowest address

byte addressability: each address identified 1 byte

endianess: byte ordering for varias with more 1 byte

little endian: IA-32: least significant byte in lowest address

big endian: most sig in lowest

Meet Pointers

What? A *pointer* variable is

- a scalar variable whose value is address
-

Why?

- for indirect access to memory.
- ↓ ↓ ↓ to function.
- common in library.
- to access memory map hardwear.

How?

→ Consider the following code:

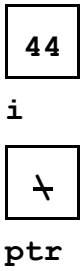
```
void someFunction(){
   int i = 44;

   int *ptr = NULL;
```

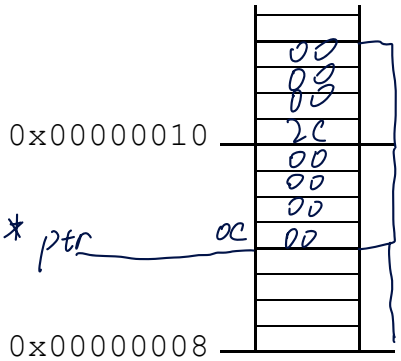
point to what type

ptr = & i.

Basic Diag.



Linear Diag.



→ What is `ptr`'s initial value? 0x0 / null address? 0x...0C type? int* size? 4 bytes

pointer: Contain adress to point

pointee: what is pointed to.

`&` *address of* operator: &i &ptr —> get address of

`*` *dereferencing* operator: *ptr.

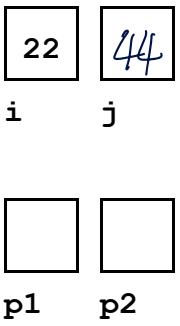
↑ only for pointer

Practice Pointers

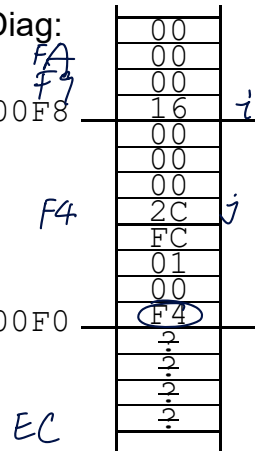
→ Complete the following diagrams and code so that they all correspond to each other:

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

Basic Diag:



Linear Diag:



→ What is `p1`'s value?

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

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

→ 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.

→ 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 compond unit of storage, elem of same type.
- access via identifier and index
- allocate as contigious fixed size block of memory

Why?

- store collection of data of same type fast.
- easier to deal than individual item for each item

How?

```
void someFunction(){
   int a[5];          // stack allocate array
```

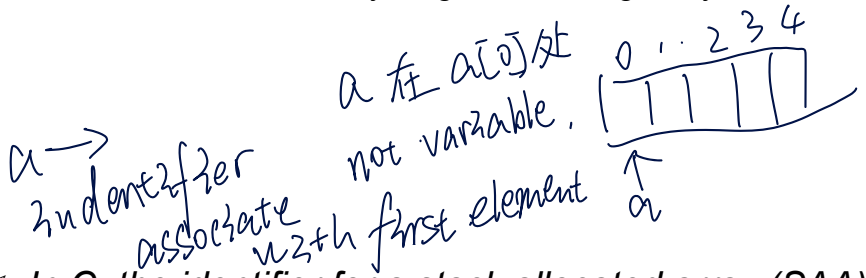
→ How many integer elements have been allocated memory? 5

→ Where in memory was the array allocation made? 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`.



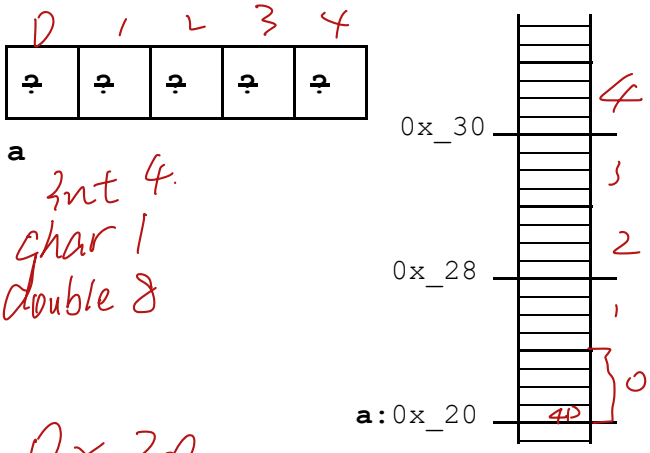
- *In C, the identifier for a stack allocated array (SAA)* not variable.
- *A SAA identifier used as a source operand* provide array address

  e.g., `printf("%p\n", a);`
- *A SAA identifier used as a destination operand* result compile error.

1D Arrays and Pointers

Given:

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



int 4.
char 1
double 8

Address Arithmetic

- `a[i]` = *(a+i)
  - int

1. compute the address

start at a's beginning 0x20.
add bytes offset to get to index i.
compile auto scale by 4

2. dereference the computed address to access the element

→ 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

Using a Pointer

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

int *p = a; // &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*

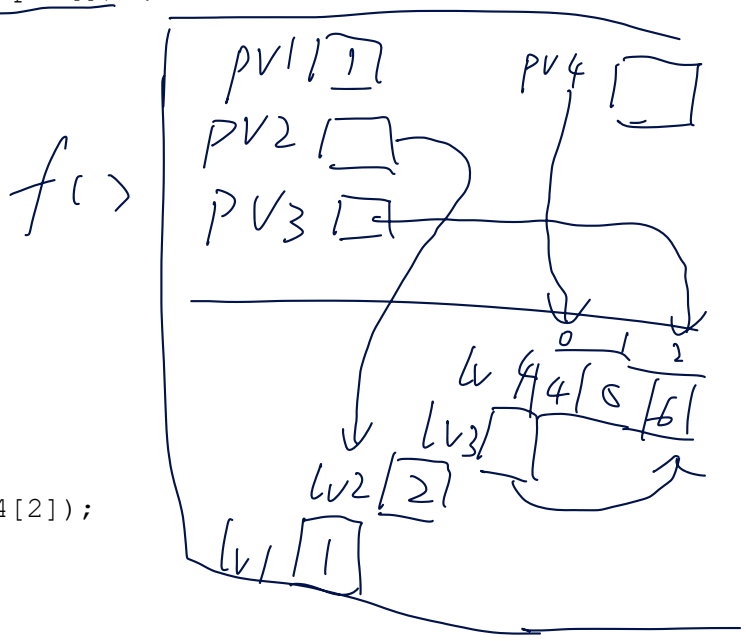
Passing Addresses

Recall *Call Stack Tracing*:

- manually trace for func call
- each func get a box (stack frame)
- top box is c r f ?

➢ 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;
}
```



Pass-by-Value

- scalars: param is a scalar variable that gets a copy of its scalar argument
- pointers: param is a ptr variable that get copy of addr
- arrays: param is a ptr variable that get copy of array address of [0]

❋ *Changing a callee's parameter* change callee's arg only

❋ *Passing an address* require caller trust callee.

∴ callee can change ptr points to

1D Arrays on the Heap

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

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

Why? Heap memory enables

- access more memory than available at compile time
- having blocks of mem allocate and free when its running.

How?

func `void* malloc(size_in_bytes)` reserve a block of heap memory of specific size. return a generic ptr that can assign to any ptr.

func `void free(void* ptr)` free heap block ptr point to.

operate `sizeof(operand)` return size

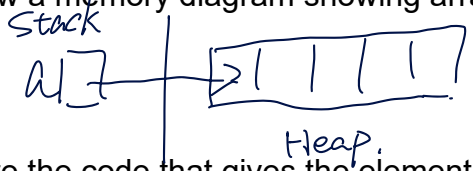
→ For IA-32 (x86), what value is returned by `sizeof(double)`? `sizeof(char)`? `sizeof(int)`?

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`.



→ 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.

```
*a = 0   // dereference
a[1] = 11  // index way
*(a+2) = 22  //
```

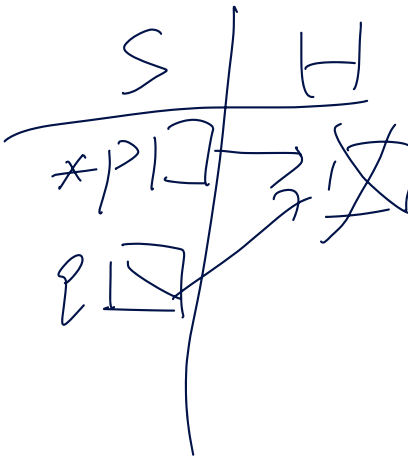
→ Write the code that uses a pointer named `p` to give the element at index 3 a value of 33.

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

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

```
free(a).
a = NULL
p = NULL.
```

Pointer Caveats



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

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

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

- *Don't dereference freed pointers!*

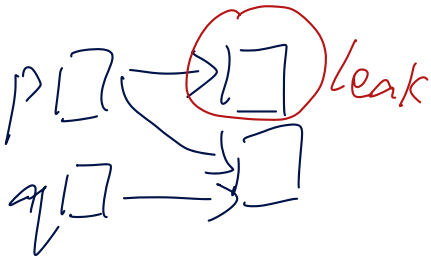
```
int *p = malloc(sizeof(int));
int *q = p;
. . .
free(p);
. . .
*q = 11;
```

dangling pointer:

- *Watch out for heap memory leaks!*

memory leak:

```
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

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

```
int *ex1() {
   int i = 11;
   return &i;
}
```

```
int *ex2(int size) {
   int a[size];
   return a;
}
```

Meet C Strings

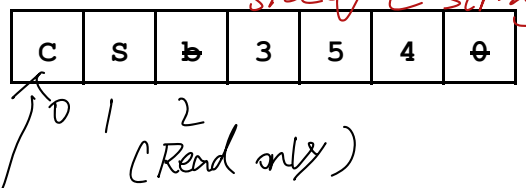
What? A *string* is

- a sequence of char terminated with null char
- a 1D array of char with string length + 1 '\0'

What? A *string literal* is "CS 354"

- a constant space code string
- allocated prior to execution

size of C strings.



❋ *In most cases, a string literal used as a source operand* provide start addr

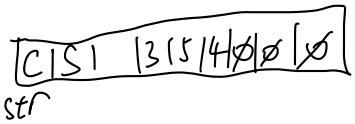
How? Initialization

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

→ Draw the memory diagram for `sptr`.

→ Draw the memory diagram for `str` below.

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



→ During execution, where is `str` allocated?

SAA

How? Assignment

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

```
sptr = "mumpsimus"; // ok

str = "folderol"; // Compiler error
```

❋ *Caveat: Assignment cannot be used*

Meet `string.h`

What? *`string.h`* is

`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* large enough including null char

buffer overflow: exceed bunds of array.

How? `strcpy`

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

```
strcpy(str, "folderol");
```

```
strcpy(str, "formication");
```
too big

```
strcpy(sptr, "vomitory");
```
SEG FAULT.

sptr = "v...." // assignment.

- *Rather than assignment, strcpy (or strncpy) must be used to* copy "c" strings from one array to another

- *Caveat: Beware of* Buffer overflow / write to code segment.