CS 354 - Machine Organization & Programming
Tuesday Sept 10th and Thursday Sept 12th, 2024

Project p1: DUE on or before Friday 9/13 available until Sunday 9/15

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

Project p2A: Released this week Friday Complete Activity A02 this week.

Homework hw1: Assigned soon, complete Activity A02 this week.

Exam Conflicts: Report for e1,e2,e3 by 9/20 : **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)

- draw basic memory diagram showing name, value, type for given code
- draw linear memory diagram showing name, address, hex contents of variable
- 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 showing stack and on heap allocations
- show byte representation of character arrays and C strings
- use **string.h** library functions: strlen, strcpy, strncpy, strcmp with string literals and C strings

This Week

`Tuesday`	`Thursday`
Finish EDIT,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   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;
```

Aspects of a Variable

identifier:

value:

type:

address:

size:

❋ *A scalar variable used as a source operand*

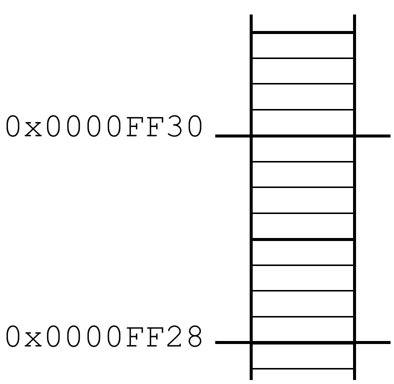
e.g., `printf("%i\n", i);`

❋ *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

byte addressability:

endianess:

little endian:

big endian:

Meet Pointers

What? A *pointer* variable is

◆

◆

Why?

◆

◆

◆

◆

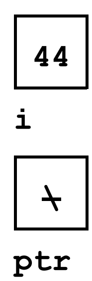
How?

→ Consider the following code:

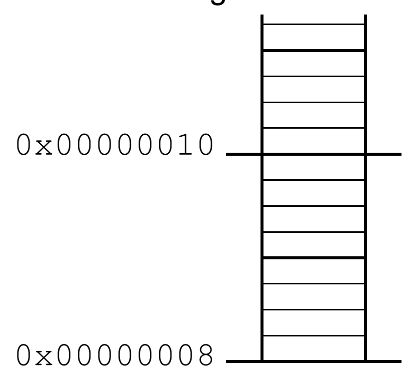
```
void someFunction(){
   int i = 44;

   int *ptr = NULL;
```

Basic Diag.



Linear Diag.



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

pointer:

pointee:

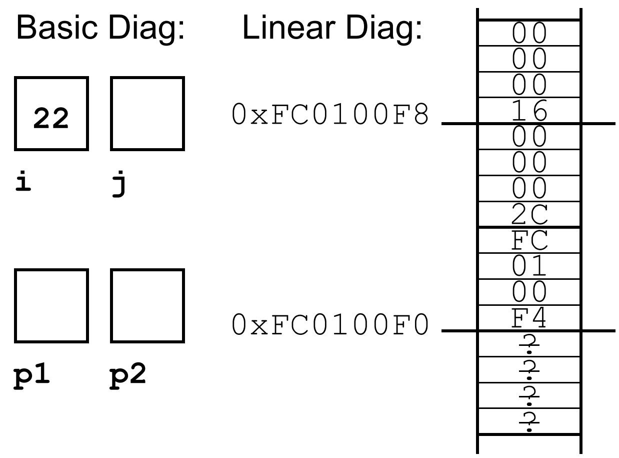
`&` *address of* operator:

`*` *dereferencing* operator:

Practice Pointers

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

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



→ 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

-

-

-

Why?

-

-

How?

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

→ How many integer elements have been allocated memory?

→ Where in memory was the array allocation made?

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

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

❋ *In C, the identifier for a stack allocated array (SAA)*

❋ *A SAA identifier used as a source operand*

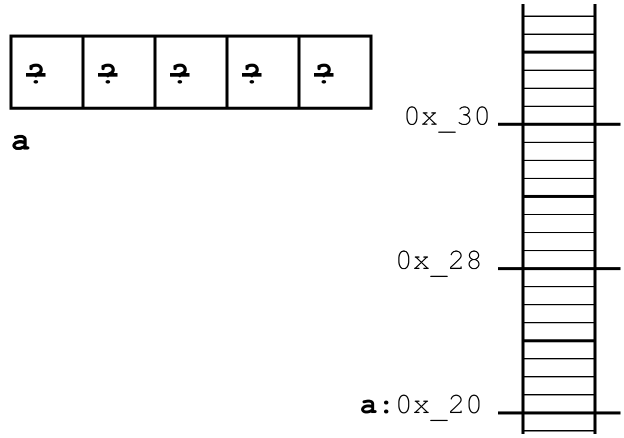
e.g., `printf("%p\n", a);`

❋ *A SAA identifier used as a destination operand*

1D Arrays and Pointers

Given:

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



Address Arithmetic

❋ *a[i]*

1. compute the address

2. dereference the computed address to access the element

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

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

Using a Pointer

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

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

❋ *In C, pointers and arrays*

Passing Addresses

Recall *Call Stack Tracing*:

-
-
-

➢ 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
- arrays: param is a

❈ *Changing a callee's parameter*

❈ *Passing an address*

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

Why? Heap memory enables

-
-

How?

```
void* malloc(size_in_bytes)
```

```
void free(void* ptr)
```

```
sizeof(operand)
```

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

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

```
void someFunction(){
```

→ 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.

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

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

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

compares nothing because it's assignment

compares values in pointers

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

-
-

What? A *string literal* is

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

-
-

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

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?

How? Assignment

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

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

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

❋ *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*

buffer overflow:

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");
```

```
strcpy(sptr, "vomitory");
```

❋ *Rather than assignment, strcpy (or strncpy) must be used to*

❋ *Caveat: Beware of*