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
- `f(void) != f()`

This Week

↑ allows any args through

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

Copyright © 2016-2023 Jim Skrentny

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: # of bytes

- *A scalar variable used as a source operand* reads the value

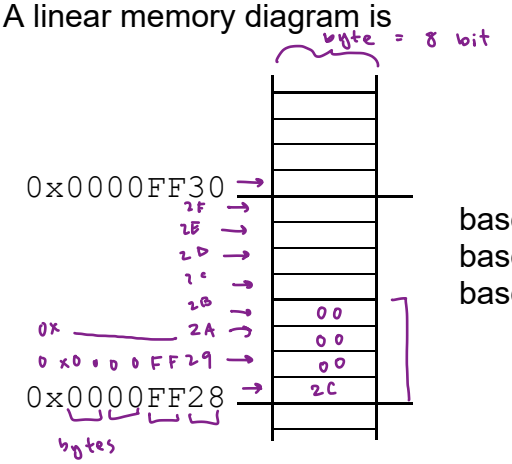
  e.g., `printf("%i\n", i);`

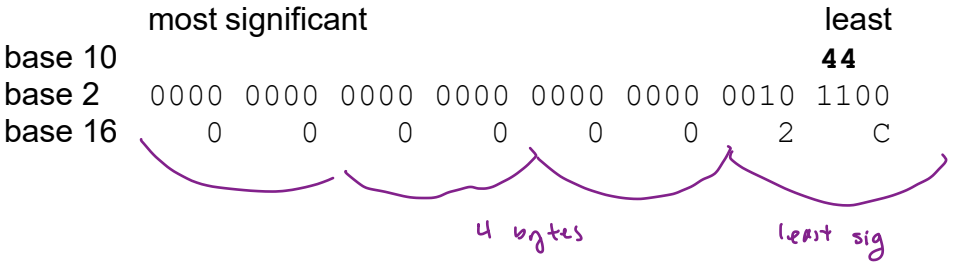
- *A scalar variable used as a destination operand* write the value to storage

  e.g., `i = 11;`

Linear Memory Diagram

A linear memory diagram is





byte addressability: each addr identifies 1 byte

endianess: byte order for variables w/ > 1 byte

little endian: (IA-32) least sig byte in the lowest addr.

big endian: most sig byte in the lowst addr.

Copyright © 2016-2023 Jim Skrentny

Meet Pointers

What? A *pointer* variable is

- a scalar var whose value is an addr.
- similar to Java reference

Why?

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

How?

→ Consider the following code:

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

   int *ptr = NULL; // 0x0
```

32 bits = 4 bytes

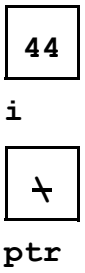
data being created

pointer

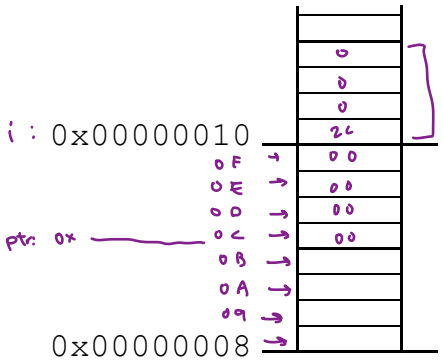
ptr = &i

my addr is an int

Basic Diag.



Linear Diag.



→ What is `ptr`'s initial value? 0x0 address? 0x_0C type? int * size? 4 bytes

pointer: contains addr, does pointing

pointee: what is pointed to

tells us addr of i

& *address of* operator: & i & ptr

* *dereferencing* operator: * ptr

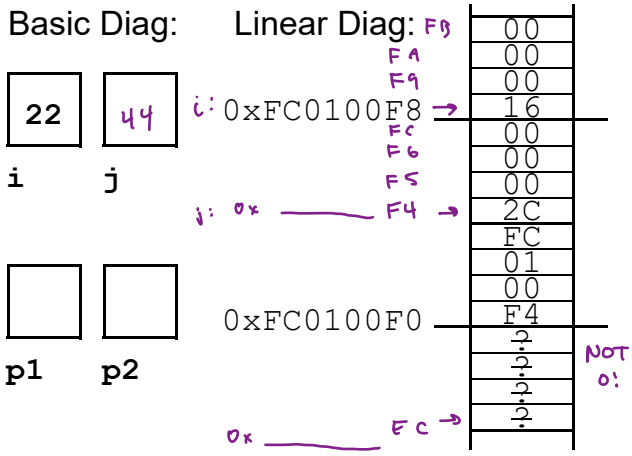
follow it and see what it points to

Copyright © 2016-2023 Jim Skrentny

Practice Pointers

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

```
void someFunction(){
   int i =  22 ;
   int j = 44;  // 22c
   int *p1 = &j;
   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;
```

Copyright © 2016-2023 Jim Skrentny

Recall 1D Arrays

What? An *array* is

- a compound unit of storage, elem of same type
- access via identifier and index
- allocated as a continuous fixed-size block of mem

Why?

- store a collection of data of same type w/ fast access
- easier to declare than indiv. items for each var

How?

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

stack allocated array

// SAA

size

Java objects are always in heap

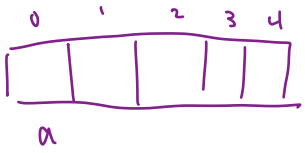
→ How many integer elements have been allocated memory? 5

→ 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)* IS NOT A VARIABLE!

✵ *A SAA identifier used as a source operand* provides array addr

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

✵ *A SAA identifier used as a destination operand* results in compiler error

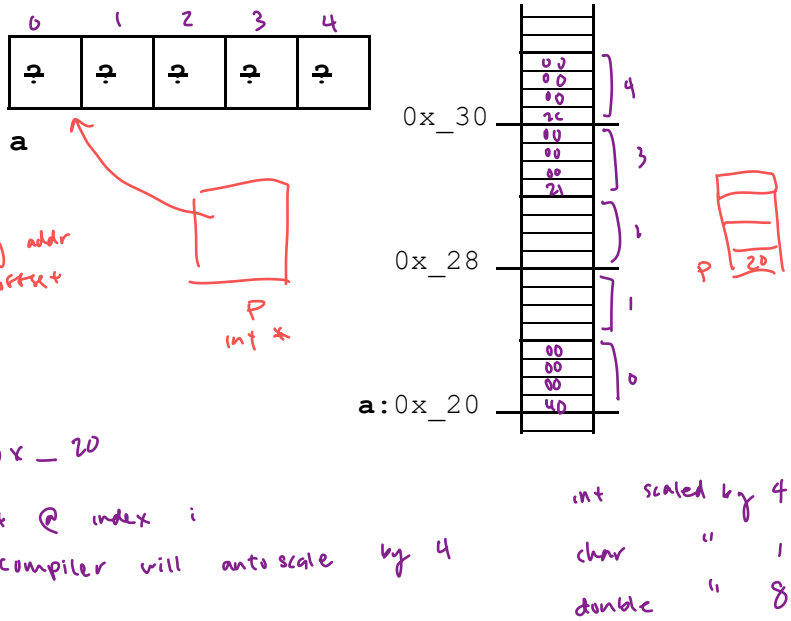
cannot reassign

Copyright © 2016-2023 Jim Skrentny

1D Arrays and Pointers

Given:

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



Address Arithmetic

❋ `a[i]` is equivalent to `*(a + i)`

starting addr + offset

int

1. compute the address

Start at a's beginning addr. 0x_20

add byte offset to get to elt @ index i

compiler will auto scale by 4

int scaled by 4
char " 1
double " 8

2. dereference the computed address to access the element `*(a + i)`

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

`*(a+3) = 33;` // @ 0x24

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

`*(a) = 77` // @ 0x20

`*(a+0)`

`*a`

Using a Pointer

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

`int * p = a;` // NOT &a

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

`*(p+4) = 44` // @ 0x2C

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

Copyright © 2016-2023 Jim Skrentny

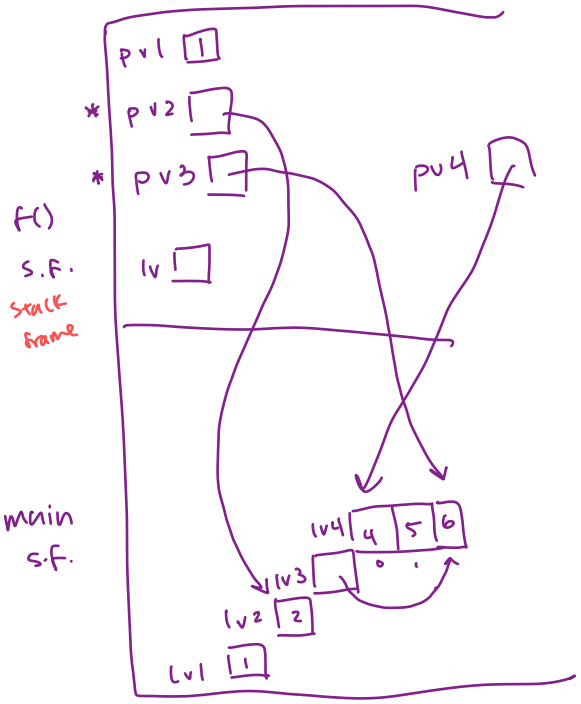
Passing Addresses

Recall *Call Stack Tracing*:

- manually trace func calls
- each func gets a box (stack frame)
- top box is the current running func

➢ 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; // lv4 is an addr, comp. auto scales
   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 var that gets a copy of addr
- arrays: param is a ptr var that gets copy of array addr of elt [0]

❋ *Changing a callee's parameter* changes the callee's args only

"safe"

❋ *Passing an address* requires the caller trust the callee

callee can modify what pointer points to

Copyright © 2016-2023 Jim Skrentny

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 allocation
"run time"

Why? Heap memory enables

- access to more memory than avail at compile time
- having blocks of mem allocated and free while prog is running

How? `#include <stdlib.h>`

func `void* malloc(size_in_bytes)`
general ptr
reserves a block of heap memory of specific size
returns a generic ptr that can be assigned to any ptr

func `void free(void* ptr)` frees heap block that ptr points to

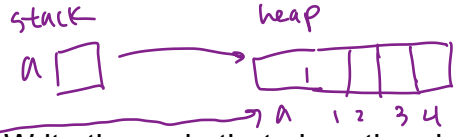
operator `sizeof(operand)` returns size in bytes of operand

→ 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`.



→ 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;
a[1] = 11;
*(a+2) = 22;
```

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

p

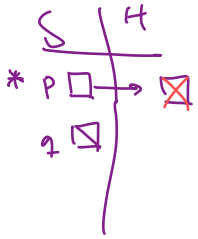
```
int * p = a;
*(p+3) = 33;
```

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

```
free(a); // makes a and p dangling ptrs
a = NULL;
p = NULL
```

Copyright © 2016-2023 Jim Skrentny

Pointer Caveats



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

```
int *p;
*p = 11;   // intermittent error
```

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

- *Don't dereference freed pointers!*

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

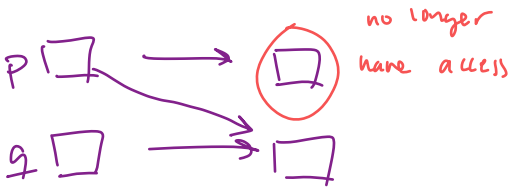
both are dangling ptr

dangling pointer: a ptr var with an addr to heap mem that was been freed

- *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;   // local var
   return &i;    // mem is not avail after func call ends
}
```

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

. to return addr of array, use the heap!

Copyright © 2016-2023 Jim Skrentny

Meet C Strings

What? A *string* is

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

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

- constant source code str
- allocated prior to execution

C	S	~~b~~	3	5	4	~~0~~
0	1	2	3	4	5	6

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

How? Initialization

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

stack

sptr

→ Draw the memory diagram for `sptr`.

→ Draw the memory diagram for `str` below.

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

0	1	2	3	4	5	6	7	8
C	S		3	5	4	\0	\0	\0

str

→ During execution, where is `str` allocated?

stack, stack allocated array

How? Assignment

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

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


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

❋ *Caveat: Assignment cannot be used*

Copyright © 2016-2023 Jim Skrentny

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* is large enough for result, including null terminating char

buffer overflow: exceeding bounds 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");   // ok

strcpy(str, "formication");   // buffer overflow

strcpy(sptr, "vomitory");   // seg fault

sptr = "vomitory";   // ok, assignment not copy
```

- *Rather than assignment, strcpy (or strncpy) must be used to* copy a "C" string from an array to another
- *Caveat: Beware of* buffer overflow to code segement