CS 354 - Machine Organization & Programming
Thursday, September 12, 2019

Waitlisted? Complete the form at: https://forms.gle/CRvL1oR8i9Bymvyo6

Jim Skrentny, 5379 CS, skrentny@cs.wisc.edu
Course website: https://canvas.wisc.edu/courses/154937

Project p1 (3%): DUE at 10 pm on Monday, September 23rd

Exam Conflicts: report any by tomorrow using form at: https://forms.gle/6TwXssFmUCh7o8GS8
TA lab consulting & PM drop-in hours: are scheduled, see links on course front page
PM BYOL: Tonight 6 pm 1207 CS
Linux Workshop: Friday 5:30 pm 1240 CS

Last Time

C Program Structure
C Logical Control Flow
Recall Variables
Meet Pointers

Today

Practice Pointers
Recall 1D Arrays
1D Arrays and Pointers
Passing Addresses

Next Time

Pointers Get More Interesting
Read:
- K&R Ch. 7.8.5: Storage Management
- K&R Ch. 5.5: Character Pointers and Functions
- K&R Ch. 5.6: Pointer Arrays; Pointers to Pointers

See: Piazza post for web alternatives to K&R readings

Practice Pointers

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

type int.

→ How many pointer variables are declared in the code above? 1.

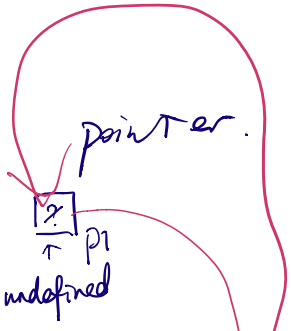
To avoid, use: int *p1, *p2;

→ What is p1's value?

no warning.

Danger. p1 is considered uninitialized

It has whatever bit pattern was in the mem allocated to it.



→ Write the code to initialize p1 with address 0.

prefer p1 = NULL;

or (p1 = 0; p1 = 0x0)

→ Given i below, write the code to make p1 point to i.

```
int i = 22;
```

p1 = &i;

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

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

↑ integer

→ What happens if the code above executes when p1 is NULL?

Danger: program crash due to segmentation fault.

→ Write the code to display p1's value?

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

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

usually not. for basic memory diagram.

→ What does the code below do?

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

Recall 1D Arrays

What? An *array* variable is

- A compound unit of storage having parts of elements whose values can change.
- Access using identifier and indexing to get to a particular element.
- allocated as a continious fixed size block of memory.

Why?

- For storing a collection of data of the same type, with fast access to its ellements.
- Because it is much easier then declaring individual variables for each items in collection.

How?

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

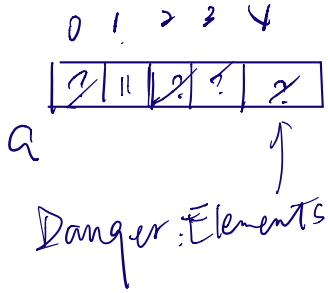
→ How many integer elements have been allocated memory? 5

→ Where in memory was the 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`.



Danger: Elements are uninitialized by default.

❋ *In C, the identifier for an array*

is associated with the starting address of the array.

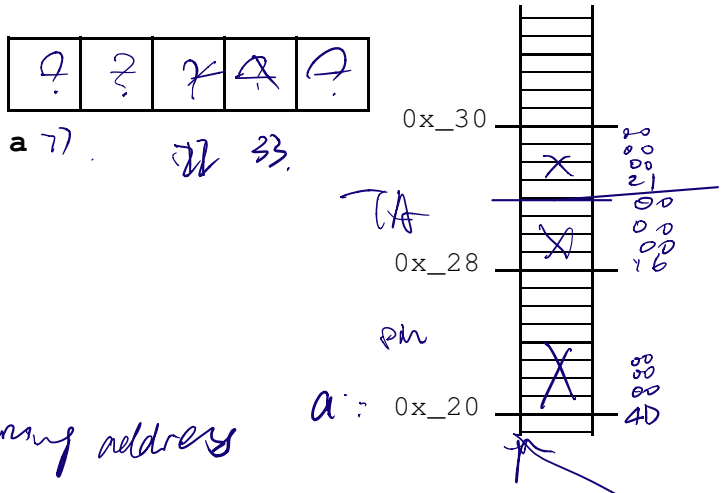
it is not a seperate var.

1D Arrays and Pointers

Given:

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

a[2] = 22;



Address Arithmetic ← Fast.

❋ `a[i]` ≡ `*(a + i)`

1. compute the address

start at array's beginning address
Add the offset to element;
which is automatically scaled by size of each element.

2. dereference the computed address to access the element

not; params are required

→ 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) =

*a = 77.

Using a Pointer

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

int *p = a

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

*(p+4) = 44;

p[4] = 44

❋ *In C, pointers and arrays* are closely related.

Passing Addresses

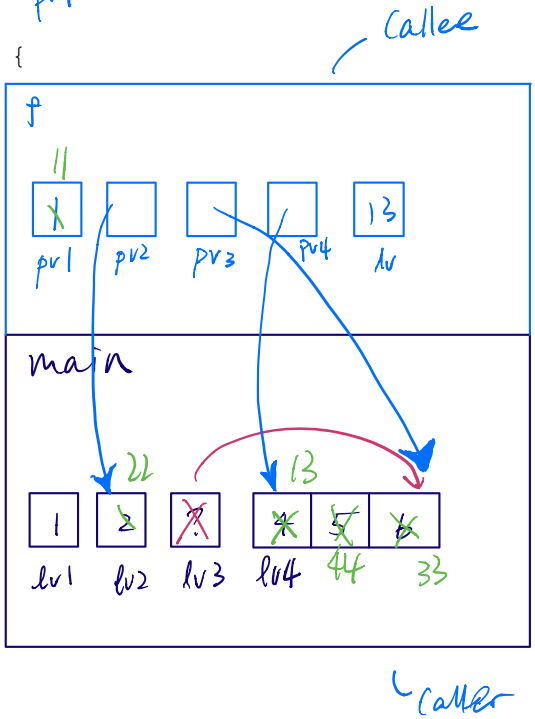
Recall *Call Stack Tracing*:

- manually excuting codes with funcs in a manner that mimic the machine.
- each function call get a box (Stack frame) which is its mem storing params, Local variables...
- Top box is running and those belows are suspended waiting for their chance 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;
}
```

≡ int *pv4



output: 1, 22, 33
13, 44, 33.

Pass-by-Value

- scalars: scalar parameter gets a copy of the argument
- pointers: pointer param gets a copy of its ADDR ARG.
- arrays: pointer param gets a copy of its array Args ADDR.

❋ *Changing a parameter*
does not change its argument. Just changes the callee's copy of the address.

❋ *BUT passing an address*
require the caller to trust the callee.
since now the callee know where the arg is and can change it.