Into to Pointers

announcements

pointer - special type of var that stores the address of amemory location

address	value	var
1012		
1013		
1014	5	X

<type> *<var name>

```
int  *ip     integer pointer
char *cp     character pointer
```

using *a gets rid of address, get value instead

& gets address of a var

Pictures are helpful!

```
int i = 7
int *p = &i
int j = *p     -> initializes value, doesn't tie together
*p = 8
```

p points to i, always

dereference p -> go to p, go to address, get value

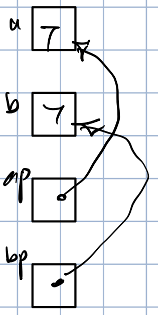
go to address, change value to 8 . doesn't change j

```
int a =7, b =7
int *ap = &a
int *bp = &b
```

a = b ✓

ap ≠ bp

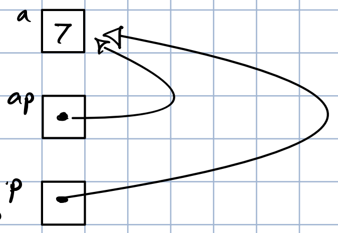
*ap = *bp follow pointer & get value



printing pointer: %d prints as hex

2 pointers are aliases if they refer to same memory location

```
int a = 7
int *ap = &a
int *p = ap
```

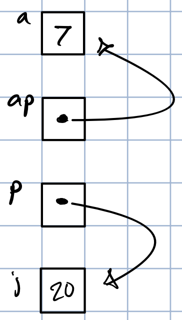


if we set *p=10, a=10, *p=10, *ap=10

```
*ap = 3
p   = &j
```

a -> 3

p = j address , now refers to new box



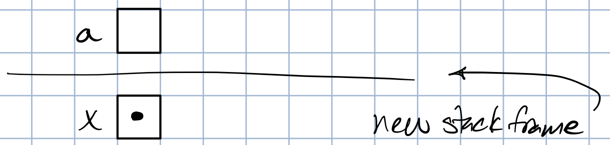
ap & p no longer aliases

recall call by value

fn's args. call by value

Use pointers to pass address of var to fn

(int *x)

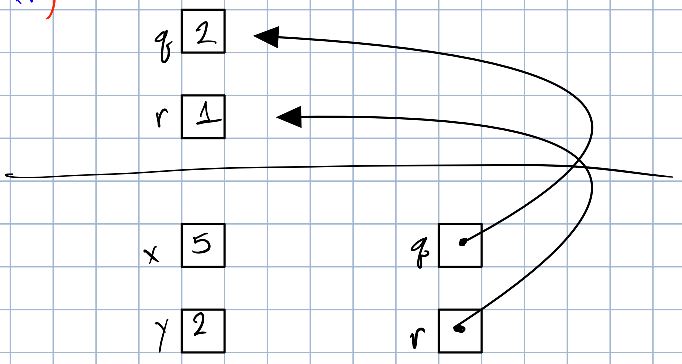


-> expect pointer to value

```
void divide(int x , int y, int *q , int *r)
    *q = x/y
    *r = x %r

int main ()
    int q, r

    divide( 5, 2, &q, &r)
```



```
void swap (int *a , int *b)
    int temp = *a
    *a = *b
    *b = temp

int main ()
    int x=5
    int y=7

    swap (&x , &y)
```

Arrays

array is data structure can be used to store values
Homogeneous
fixed length
continuous & sequential in memory

```
int    a[5]                                  int array w/o values
double b[] = {1.1, 2.2, 3.3, 4.4, 5.5}       initialized w/values

double x = b[1]          x = 2.2             0 based indexing
a[0]     = 1             change value        mutable elements
└▷ 'l-value'
```

C has no check bounds

segmentation fault: "hey you're writing somewhere in memory you shouldn't be!"

array variable actually just pointer to location of 1st element

```
int c = *b
int d = b+2              & b[2]
int e = *(b+2)             b[2]
```

arrays as fn parameters
passes beginning of array   not easy to get array length

```
double sum( double a[], int len)
    double rv = 0
    for ( int i = 0 ; i < len ; i++)
        rv += a[i]
    return rv
```

return an array

```
      ↳ return pointer
double*  square ( double a[ ], int len)
    double  rv[len]
    for ( int i = 0 ; i < len ; i++)
        rv [i] = a[i] * a[i]

    return  rv
```

Wrong

don't return addresseses in the stack, it goes away

need to allocate data to heap → special area of memory for dynamically stored values
use malloc from stdlib.h

```
          ┌─▷ pointer w/var                 ┌─▷ returns void*
double   *cp =   (double*)   malloc( len *  size of(double))
```

make sure to get length correctly

always check return value

```
if ( rv == NULL )
        "not able to allocate for rv"
    exit(1)
```

need to deallocate heap memory

```
free( cp )
```

→ when you're done

every call to malloc should have matching call to free

```
char* cp = (char*) malloc (sizeof array)
if ( cp == NULL )
    ...
use cp
free( cp )
```

strings!

char type used to store character

strings are stored as arrays of characters w/sentinel '\0' to mark end

```
char s[] = "Hello"
                 'H' 'e' 'l' 'l' 'o' '\0'
```

same indexing & updating as arrays
    special syntax for string literals

need to include null terminator for string

```
char s2 = (char*) malloc( size of char (char) * 4)
s2[0] = 'H'
s2[1] = 'e'
s2[2] = 'y'
s2[3] = '\0'
```

Malloc

for SE3: don't forget null terminator '\0', allocate enough space for word + null, memory isn't default zero

only free once done

calloc(size , sizeof(int)) → allocate memory to zero

malloc(size * sizeof(int)) → allocate memory

char type used to store character or small numbers

char c = 123

char d = 'd'

man ascii in terminal

to print a char: %c , its int manipulation: %d

strings are stored as arrays of chars w/sentinel

\0 written as 2 characters, represent 1

can use malloc

```
char* s2 = (char*) malloc( sizeof(char) * 4)
s2[0] = 'H'
s2[1] = 'e'
s2[2] = 'y'
s2[3] = '\0'
```

special syntax for string literals

arrays (so strings too) are mutable

%s print specifier

library string.h for working w/strings

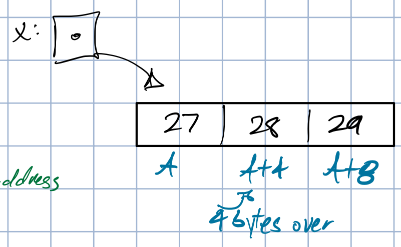
strlen(s) → length w/o null. # of chars represented

strcamp("a","b") if "a" == "b" in lexicode, 0

if "a" > "b" .

if "a" < "b" .

int x[] = {27, 28, 29}



x[0] → 27 value

* x → 27 value

x + 1 → next element , 28 address

& x[1] → 28 address

*(x+1) → 28 value

```
strlen( char*   s)
    char*   curr
    while ( *curr  ! =  '\0')
        curr++

    return   curr-s
```

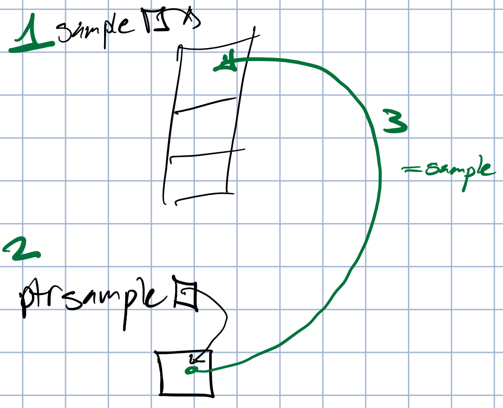
works b/c going to next element
pointer arithmetic

```
for ( char *curr=s ; *curr++ ; ){
    ;
}
```

```
                 1
char   *sample[n]
char   ** ptr_sample = sample
           2            3
```

→ allocate array of size n
allocate to array of strings

```
ptr_sample [0]   =  "hello"
ptr_sample [1]   =  "world"
```



Struct

struct – user defined datatype in C

```
struct   <struct_name>{
    <fields>;
}
```

```
struct   point{
    double x ;
    double y ;
}
```

```
struct   <struct_name>   <var_name>
struct    point          p1   =  {1.0, 2.0}
```

allocate memory
curly braces only for initializing

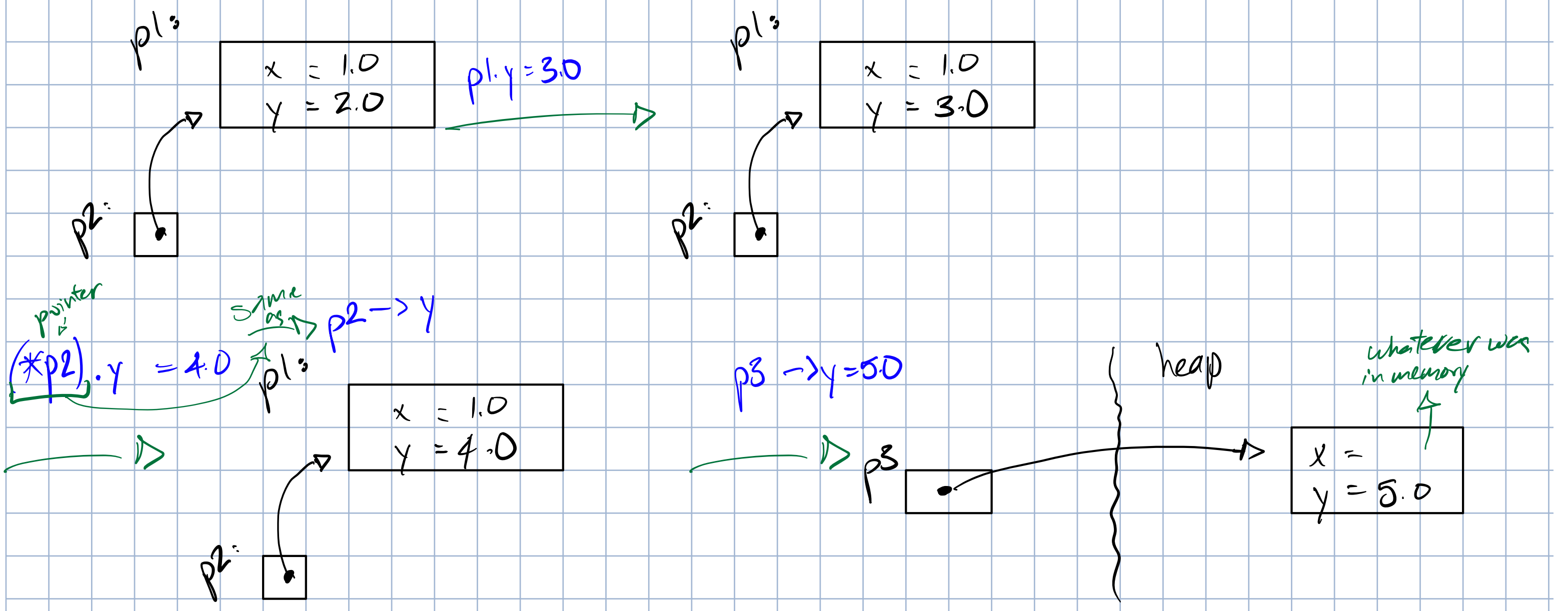
can use dot notation

L value p1.x = 1.1   can update value

can be used anywhere
- local var
- param type
- return type (copy value)
- array elements
- targets of pointers
- struct field types

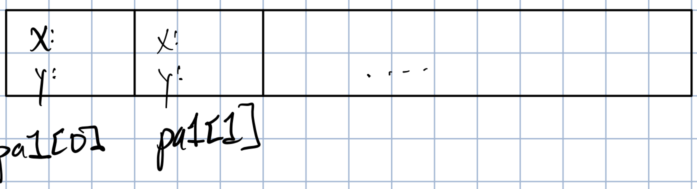
fn's call by value, not reference

```
struct   point*   p2  =  &p1          alias for p1
struct   point*   p3  =  (struct   point *) malloc(sizeof(struct point))
```



arrays of structs

```
struct   point   pa1[5]
struct   point * pa2  =  (struct point *) malloc( sizeof(struct point) *8)
pa1[1]= midpoint (p1,p2)
double   x   =   pa1[1].x
```



array of struct pointers

```
struct   point   *ps1[5]          array of struct point *
struct   point   **ps2 = (struct point **) malloc(sizeof(struct point*) *8)
```

ps1	struct point * *	
ps1[1]	struct point *	
*ps1[1]	struct point	dereference. follow address & get value
(*ps1[1]).x	double	ps[1] -> x

Multidimensional Arrays

```
<type>   <name> [size 1] [size 2] ... [size N]

double   a[5][5]
int      b[2][3] = { {0, 1, 2},
                     {3, 4, 5}} ;
```

Set of curly braces for each row

bracket notation to accesselement

`b[0][1] = 6`

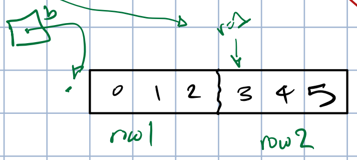
element i of array is e location

`a+i = a + i*sizeof(double)`   same as bracket

move over i doubles
address arithmetic

byte arithmetic

$b[r][c] = *(b + r \cdot 3 + c \cdot 1)$ pointer/address arithmetic



C needs to know how much needed for each row. How many columns? missing

```
void   print_array ( int rows,  int cols,   int  a [rows][cols] )
   for ( i=0     i<rows     i++ )
      for( j=0    j<cols     j++ )
         printf( " %d ", a[i][j])
      printf( "\n" )
```

not needed

but need it, so good practice is to use all

for n, need n-1 dims.

dynamically allocated!
just an array of pointers

```
int rows=3, cols = 4
int ** a =      (int **) malloc ( rows * sizeof (int *) )

for ( int r=0      r < rows      r++ )
   a[r] = (int *) malloc( cols * sizeof(int) )
   for ( int c=0      c < cols      c++ )
```

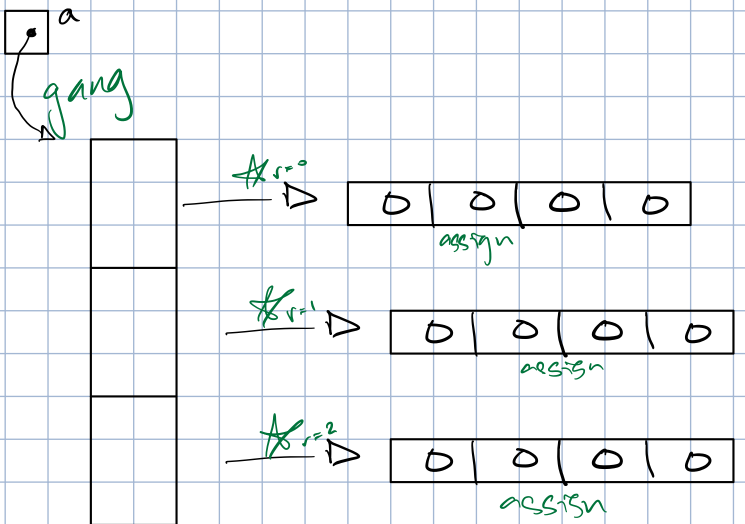
you're allocating pointers

gang

you're allocating ints

a[r][c] = 0 assign

T * name= (T *)malloc(sizeof(T) * rows)



free each row explicitly , then free array holding pointers

```
for ( i=0   i<rows   i++   )
        free(a[i])
free( a )
```

check malloc

```
void   * ck_malloc ( size_t   num_bytes ,
         void  *star   =    malloc ( size_t   )
```

didn't get all

can make library

header file w/ declarations

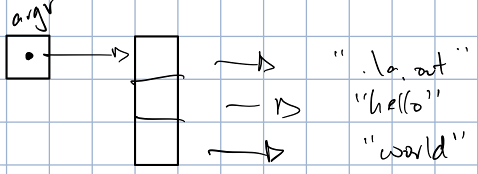
make a seperate .c file using it # include header file

command line args

```
int   argc ,  char*
```

argc → amountat

argv [0] → name of file



argc → # of command line args including param name

argv → array of strings where each string is command arg. (including name)

Union

struct - user defined data type used to bundle items into a single type

```
struct circle
    struct point center
    double radius

struct square
    struct point topleft
    double side
```

AND

can we write single fn to calculate area?

union - user defined data type that allows different data types to be stored in same location
only 1 member can contain a value

```
union shape {
    struct circle c;
    struct square s;
};
```

OR

make a square or circle

```
union shape s1
s1.c.center = p1
s1.c.radius = 1.0
```

```
union data {
    int x
    double y
}
union data d1
d1.x =5
union data *d2
d2 -> y =5.5
```

allocates space for largest member

allocate pointer

can still access unused members, will give garbage value

```
struct tagged_shape
    union shape shape
    enum shape_tag tag
```

enum shape_tag = {CIRCLE, SQUARE}

```
struct tagged_shape s1
s1.shape.c.center.x = 1
s1.shape.c.center.y = 2
s1.shape.c.radius = 5
s1.tag = CIRCLE
```

```
struct tagged_shape s2
s2.shape.s.topleft.x = -1
s2.shape.s.topleft.y = -1
s2.shape.s.side = 2
s2.tag = SQUARE
```

```
double area(struct tagged_shape s)
    switch(s.tag)
        case CIRCLE
            return πr^2
        case SQUARE
            return s^2
```

typedef can rename any type

Debugging

where does it segfault?

$$\text{clang } \underbrace{\text{-g debugging-ex.c}}_{\text{add info}} \; \underbrace{\text{-o debugging.ex}}_{\text{name of output file}} \quad \textbf{no .c}$$

1 window for compiling → term1
" " debugging → term2
" " editing

term2

lldb goes into debugger

file debugging_example this is program to debug

run run program @ main & see what happens

say it fails, where it failed the specific line

p int_ptr print variable & amt.

running a recompiled file kills current process

term2

thread backtrace -c 5 shows 5 frames
help show infinite recursion

set a breakpoint

breakpoint set --file debugging-ex.c --name fact

where f^n is at

name of f^n to break @

breakpoint list — all breakpoints

frame info — wya, info

frame variable — variables rn

step
s
→ either way, step one line @ a time
execute next step. will step into f^n

next — doesn't step into f^n

breakpoint set --file debugging-ex.c --line 18

breakpoint stops @ another breakpoint

continue

lldb → really good @ seg fault

now look @ valgrind!

term

valgrind mem_issues — want to see 0 errors

says whether there are mem allocation issues

can say if there is unfreed memory

valgrind --leak-check=full mem_issues

says read of info on freed memory

lldb didn't fail, but valgrind brings it up