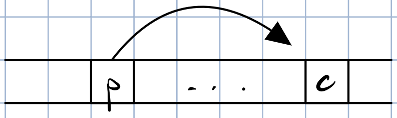
terminal vars
code output

pointer - variable containing address of a variable
closely related w/arrays

Pointers and Addresses

memory usually stored in consecutively numbered/addressed cells

if p is a pointer pointing at c :



unary operator & gives address of object only works w/objects in memory

p = &c $\rightarrow$ assigns address of c to p

unary operator * is indirection / dereferencing operator, accesses the object

```
int x=1, y=2, z[10];
int *ip;                ip is pointer to int

ip = &x                 ip points to x
y = *ip                 y is now 1
*ip = 0                 x is now 0
ip = &z[0]              ip points to z[0]
```

```
double *dp, atof(char *)
```
expressions *dp & atof(s) have values of type double
arg in atof is a char pointer

```
y = *ip + 1
```
takes whatever *ip points@, add one, assign to y

b/c pointers are vars, can use w/o dereferencing iq = ip $\rightarrow$ iq points to what ip points to

Pointers and Function Arguments

b/c C fn call by value, no direct way to alter variable in calling fn

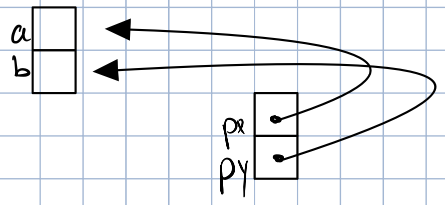
```
swap(a,b)        void swap(int x, int y)
                   int temp = x
                   x = y
                   y = temp
```
wrong, doesn't affect args a & b, only swaps copies of a & b

swap (&a, &b)
b/c & B an address,
pass thru pointers &a & &b

```
void swap(int *px, int *py)
    int temp = *px
    *px = *py
    *py = temp
```

params are pointers

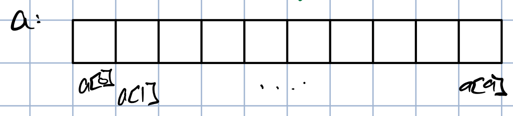
pointer args allow f^n to access & change objects in f^n



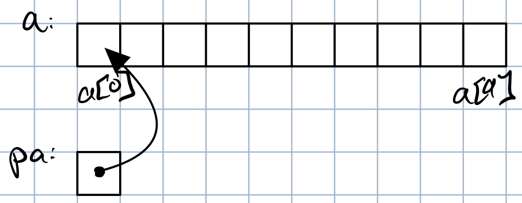
Pointers and Arrays

any operation achieved w/ array subscripting can be done w/ pointers

int a[10] defines array of size 10

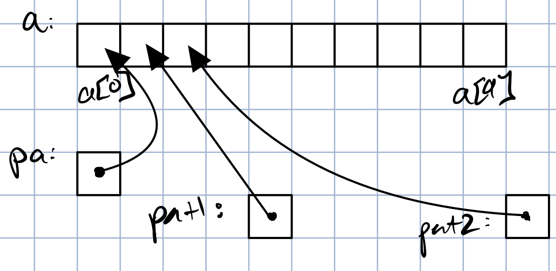


int *pa declared pointer
pa = &a[0] set pa to point a[0]



x = *pa copy contents of *pa to x. x=a[0]'s content

*(pa+1) points one ahead of pa



"adding 1 to a pointer" is that pa+1 points to next object

value of var type array is address of element zero of array
b/c array name is synonym for location of initial element

pa = & a[0] $\leftrightarrow$ same
pa = a

initial zeroth element address
add one

b/c of this, reference to a[i] is equivalent to writing *(a + i)

$\rightsquigarrow$ & a[i] & a+i are identical
$\rightarrow$ address of ith element beyond a

array & index equivalent to pointer & offset

a pointer is a variable: pa=a ✓ pa++ ✓
an array name is not a variable: a = pa ✗ a ++ ✓

when array passed thru fn, what is passed is location of initial element
with fn. arg is a local variable $\rightarrow$ array name param is pointer (var w/address)

```
                      → pointer
int   strlen( char *s)
  int n
                    → endoffile
  for ( n=0 ; *s != '\0' ;   s++      )
    n++

  return n
```

b/c s is a pointer, perfectly legal to increment it

s++ no effect on char string in fn, increments private copy of pointer

```
strlen( "hello world" )     string constant
strlen(   array       )     char array[100]
strlen(   ptr         )     char *ptr
```
} all legal

as formal parameters, in fn defn, char s[] $\xleftrightarrow{\text{equivalent}}$ char *s

can pass part of an array to a fn by passing pointer to beginning of subarray
f(& a[2])   f(a+2)

within fn, param declaration: f(int a[]) {..}   f(int *a){...}

can index backwards if elements exist   p[-1]

Address Arithmetic

p++ increments pointer to next element

p+=i increment to i elements beyond current position

can create a stack structure last in, first out

can set pointer to 0, standing for NULL, no other integer can be used

if pointers p & q point to same array, relations work
$\rightarrow$ ==, !=, >, <=
" " , can use subtraction
q-p+1 is # of elements, inclusive

```
int strlen(char *s)
    char *p =s

    while (*p != '\0')
        p++                 → next character

    return p-s              # of characters advanced
```

pointer manipulations take into account size of object pointed to (char, int, ...)

valid:
1. assignment of pointers of same type
2. adding or subtracting a pointer & an integer   p+=1
3. subtracting or comparing 2 pointers to members of same array   p-s
4. assigning or comparing to zero   p==0

Character Pointers and Functions

string constant is array of chars "Howdy!"
array is terminated by: \0
accessed by pointer to first element

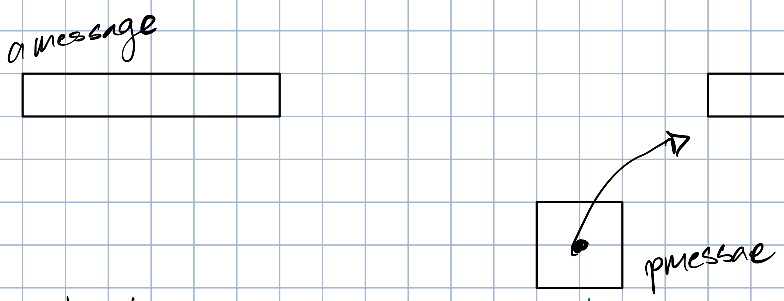
```
char *pmessage

*pmessage = "Howdy!"     → assigns pointer to char array   (not a copy)
```

in C, we can't process as one unit

char amessage[] = "Howdy!" array holds sequence of chars & /0
indiv. chars can be changed.
amessage refers to same storage

char *pmessage = "Howdy!" pointer initialized to point to string constant
pmessage can be made to point elsewhere



copy t to s

pointer to s
pointer to t

```
void strcpy(char *s, char *t)
    while ( (*s =*t) != '\0' )
        s++
        t++
```

assigns pointer *s to value of *t

can also just do

```
while(*s++ = *t ++)
    ;
```

increment . if \0 , it'll be false

compare strings

if s<t → <0
s==t → 0
s>t → >0

```
int strcomp(char *s, char *t)
    for ( ; *s == *t ; s++ , t++)
        if (*s == '\0')
            return 0
    return *s - *t
```

standard push pop
*p++ = val
val = *--p

Pointer Arrays; Pointers to Pointers
b/c pointers are vars, they can be stored in arrays like other variables

for strings, each can be accessed by pointer to first character

these pointers can be stored in arrays

MultiDimensional Arrays

```
static char daytab[2][13] = {
        { , , , . . - - },
        { }
}
```

13 elements

to index, need 2 [ ] , not [,]
[row] [column]

to have 2-D array as parameter, need to specify column amt

```
f(int daytab[][13])   <-same->   f(int daytab[2][13])
          <-same->   f(int (*daytime)[13])
```

parameter is pointer to array w/13 integers

Initialization of Pointer Arrays

```
char *month_name(int n)
    static char *name = {
        "January", ....        }
    return name[n]
```

return pointer to string

array of character pointers

characters of ith string placed somewhere
pointer to string stored in n[i]

Pointers vs Multidimensional Arrays

```
int a[10][20]
int *b[10]
```

a[3][4] & b[3][4] synatically same refferring to some int
but a is a 2-d array w/ 200 elements
b only allocates 10 pointers. each of these can point to any # array, not necessarily 20

Command line Arguments

when main is called, it's called w/2 arguments
1. argc — the # of command-line args program was invoked with
2. argv — pointer to an array of strings that contain the arguments. one per string

simplest: echo hello world
conventionally, argv[0] is name of program → argc at least 1
if argc is 1 → no more command-line programs after program name

here, argc is 3 →

argv[0] → echo
argv[1] → hello
argv[2] → world

require argv[argc]
to be null pointer

++argv

b/c argv is pointer to beginning of array of arg strings, incrementing by 1
points it to original array argv[1] instead of name argv[0]

*argv is then the pointer to the argument

optional flags or parameters begins w/a minus sign