# October 31

## Arrays of Pointers

**Example 1**

```
char *a[] = {"hello", "world"};
char *p = "hello";
printf("%s", a[0]);     /* hello */
printf("%c", a[0][0]);  /* h */
```

a[0] and a[1] are both char

*NOTICE: essentially a two dimensional array*

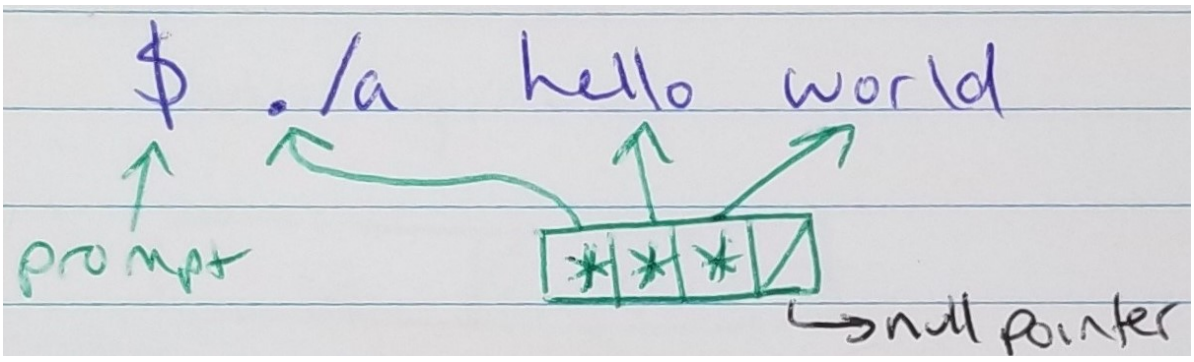
*NOTE: the print function will start at a[0] and keep printing until the '\0' is reached. In order to print one character we need to use "%c"*

**Example 2**

```
int main(int argc, char * argv[]) {...}

/* same as */

int main(int argc, char ** argv) {...}
```

*DISTINCTION: the first is an ARRAY OF POINTERS, the second points to an array of pointers, that's why the missing [ ]*

*NOTE: This is the standard command line arguments format we're just explaining what happens at a deeper level**




The shell creates an array of pointers pointing to copies of the command line arguments and append a null pointer and passes that array into argv

1st creates a pointer to string "./a"

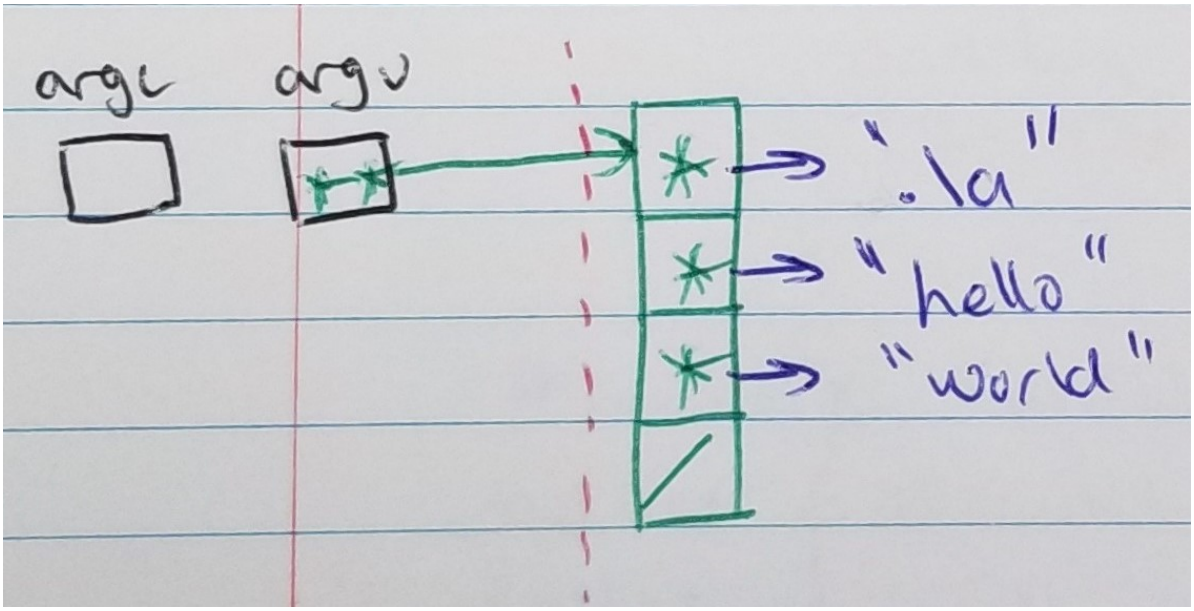
2nd creates a pointer to string "hello"

3rd creates a pointer to string "world"

4th is a null pointer

argc = 3

*NOTE: Inside main, argv is a double pointer that points to the array of pointers outside of main*



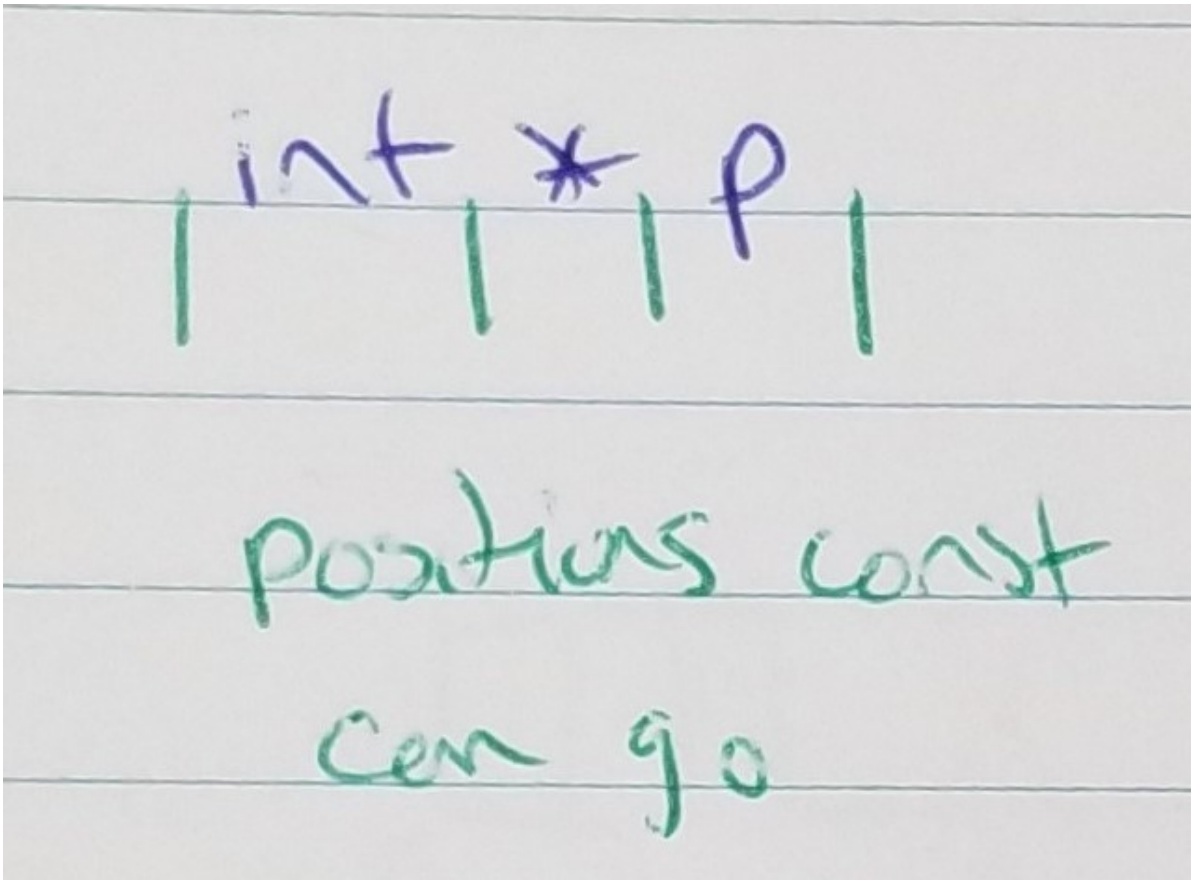

## Modifier 'const'

*IMPORTANT REMINDER:*

A const can only be initialized, it cannot be assigned to

```
const int n = 1;     /* OK; initialization */
n = 2;               /* invalid  --> it's a constant */
```

Using const with pointers:




```
const int *p
int const *p
int *const p
int *p const    /* invalid */
```

*NOTE: also cannot have two const* in same statement

Read from right to left

```
const int *p    /* p is a pointer to integer constant */
int const *p    /* p is a pointer to constant integer */
```

these are distinct in that the pointer is constant

both can be used interchangeably

you can change the pointer, but you can't use the pointer to change the integer it points to

```
int * const p   /* p is a constant pointer to integer */
```

in this different case the constant qualifies the pointer

we can't change the pointer, but we can change the integer it points to

```
int * p const;      /* INVALID */
```

this is a meaningless statement

**Examples**

```
int n = 123;
const int *p= &n;
int *const q = &n;

*p = 456;   /* INVALID --> *p is a constant */
*q = 456;   /* OK, changes n to 456 */
q = p;      /* INVALID, q is a const */
p = q;      /* OK */
```

```
const int * const r = &n; /* VALID BUT... */
```

both r and *r are const

we can't change r, and we can't assign to *r

# Structures

Allow us to package data (possibly of different types) together

*CONSIDER: we can't group an int and char together*

**Example 1: struct definition**

struct is the key word here

```
struct grade {      /* defines the structure type */
    char id[10];    /* members of the structure*/
    float score;
};
```

'grade' is a structure definition, it defines the structure type

id and score are members

```
struct grade g;
int n;
g.score = 60;
strcpy(g.id, "a12345678");
printf("%s: %.2f\n", g.id, g.score);    /* a12345678: 60.00 */
```

```
g is the variable name, struct grade is the type
```

just as n is the variable name, and int is the type

we must use strcpy to print to a string within a structure

a simple printf statement can access and print out contents of a structure

*REMINDER: strcpy(location, string)*

*COMPARISON: we need to create variables of this struct type,  like we create variable a of type Animal*

*NOTE: we use dot notation to access the members of a struct*

**Example 2: array of structs**

```
/* references Example 1 */

struct grade a[102];    /* creates an array of grade structures */

strcpy(a[0].id,"a00000666");    /* populating index 1 */
a[0].score = 49.75;

strcpy(a[1].id, "a11111111");   /* populating index 2 */
a[1].score = 99.75
```

Assuming we have stored all 102 grades:

```
for (i=0; i < 102; i++)
    printf("%s: %.2f\n, a[i].id, a[i].score);
```

if not fully populated, would return nonsense data already in the memory block reserved when array created

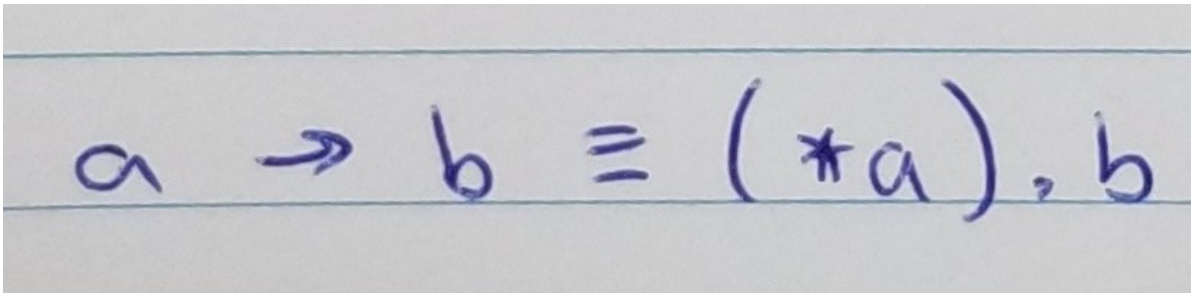
**Example 3: Accessing data members via pointer?/**

```
/* references Examples 1 & 2 */

struct grade g;
struct grade *p = &g;
(*)p.score = 89.25;     /* p -> score = 89.25 */
strcpy((*p).id, "a22222222");
```

*CRITICAL: another notation for ( * )p.score -> p -> score*

So, when you create a pointer to a structure, you'd ordinarily have to reference it through dot notation through a pointer, as in (*)p.score, but you can more easily simply use p -> score to reference the structure content data

*NOTE: you need the brackets because dot reference has a very high precedence in C, but * has very low precedence*

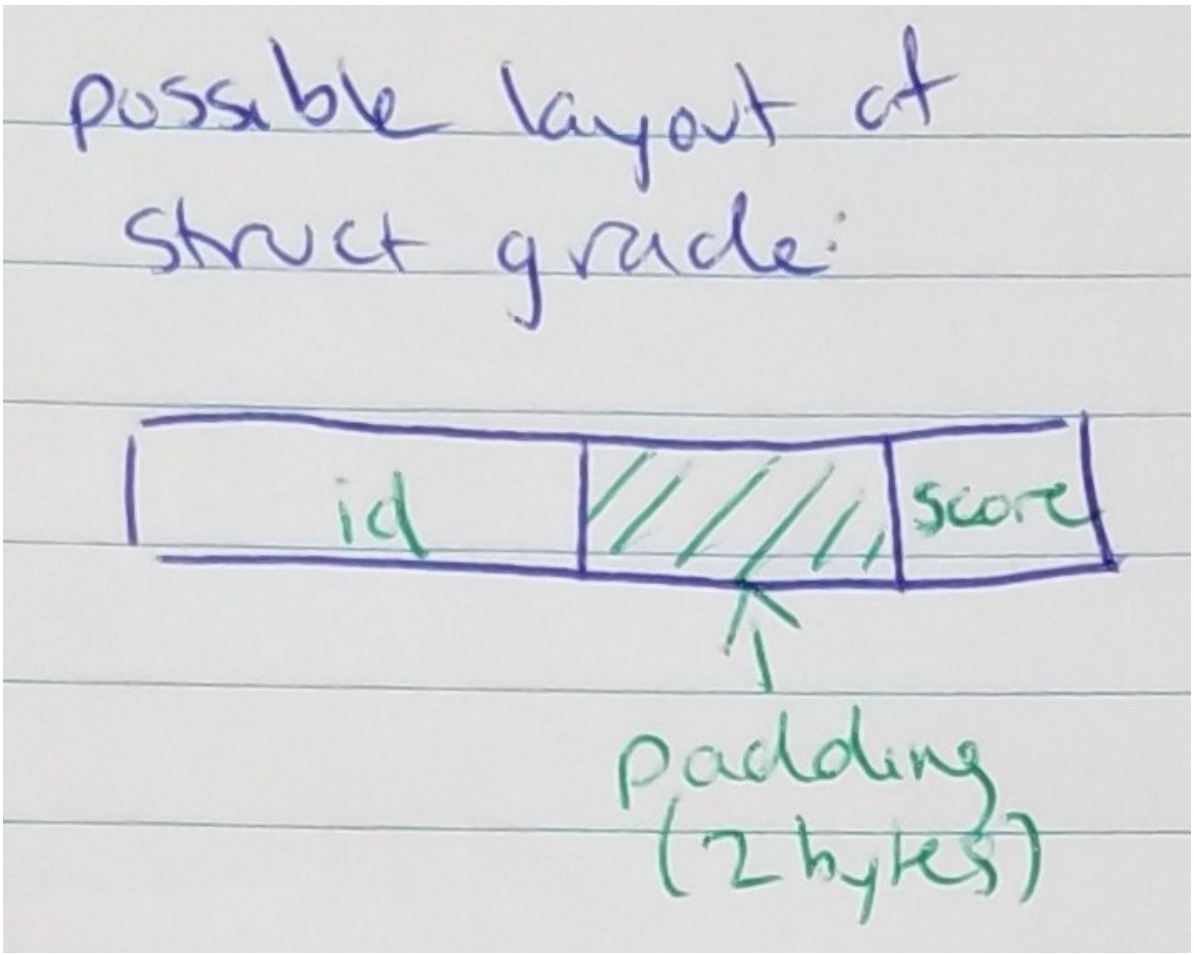



## What is the size of a struct grade?

Assume 4-byte floats

```
/* references Example 1 */

sizeof(struct grade) >= 10 + 4 = 14 bytes
```

*NOTE: the compiler is allowed to add padding between members of a structure and at the end of a structure due to alignment requirement*




In general, the size of a structure is bigger than or equal to the sum of the sizes of its members

This is not always the case, the compiler will add padding bytes

It's for alignment. Many processors can't access 2- and 4-byte quantities (e.g. ints and long ints) if they're crammed in every-which-way.

Suppose you have this structure:

```
struct {
    char a[3];
    short int b;
    long int c;
    char d[3];
};
```

Now, you might think that it ought to be possible to pack this structure into memory like this:

```
+-------+-------+-------+-------+
|           a           |   b   |
+-------+-------+-------+-------+
|   b   |           c           |
+-------+-------+-------+-------+
|   c   |           d           |
+-------+-------+-------+-------+
```

But it's much, much easier on the processor if the compiler arranges it like this:

```
+-------+-------+-------+
|           a           |
+-------+-------+-------+
|       b       |
+-------+-------+-------+-------+
|               c               |
+-------+-------+-------+-------+
|           d           |
+-------+-------+-------+
```

In the packed version, notice how it's at least a little bit hard for you and me to see how the b and c fields wrap around? In a nutshell, it's hard for the processor, too. Therefore, most compilers will pad the structure (as if with extra, invisible fields) like this:

```
+-------+-------+-------+-------+
|           a           | pad1  |
+-------+-------+-------+-------+
|       b       |     pad2      |
+-------+-------+-------+-------+
|               c               |
+-------+-------+-------+-------+
|           d           | pad3  |
+-------+-------+-------+-------+
```

# Nested Structures

```
#define NAMESIZE 20
#define IDSIZE 10

const int n = 10;   /* magic number */
int a[n];
```

this kind of magic number declaration to instantiate an array is not allowed in ANSI C, so we use #define to avoid magic numbers

using variables to allocate memory for arrays is not allowed

```
struct name {
```

```
    char first[NAMESIZE];
    char last[NAMESIZE];
};

struct name n;
strcpy(n.first, "homer");
strcpy(n.first, "simpson");

struct record {
    char id [IDSIZE];
    struct name name;
    float score;
}

struct record comp2510[102];
compt2510[0].score = 65.0;
strcpy(comp2510[0].id, "a66666666");
strcpy(comp2510[0].name.first, "perry");
strcpy(comp2510[0].name.last, "x");

void print_record(const struct record *p) {
    printf("%s : %s, %s : %.2f\n", p->id, p->name.last, p->name.first,      p-
>score);
}

print_record(&COMP2510[0]);
    /* a66666666: x, perry: 65.00 */
```

*NOTE: for the record, to do something like the print_record function, you MUST use a pointer and call it's location when you call the function*

*otherwise you pass by value, and you want to pass by reference*

*NOTE: using const struct record means p cannot be used to access a record*

To eliminate using the word struct repeatedly, we can assign a struct  a name

# Modifier 'typeDef'

A keyword used to give a type another name

```
int integer;    /* integer is name of a variable*/
```

create a variable named integer that has type int

```
typedef int integer;    /*integer is name of type */
```

we have given 'int' another name

integer is the same as type int

**Example: Give the type "array of 25 ints" the name "set"**

after the typedef, we can do this:

```
int set [25];           /* set is an array of 25 int */
typedef int set [25]    /* set is now of the type |array of 25 ints| */

set a, b, c, d, e;      /* creates int[25] on each instantiation */
a[0] = 55; d[12] = 66;  /* which can then be called by index */
```

Create a variable of the type you want, int set[25]

now put keyword typedef in front of it and instead of the set being a variable name, it becomes the name of a type

We have seen:

```
#define NAMESIZE 20
#define IDSIZE 20

struct name {
    char first[NAMESIZE];
    char last[NAMESIZE];
};

struct record {
    char id[IDSIZE];
    struct name name;
    float score;
};

typedef struct record record;
record comp2510[102];
```

But another way of doing this is:

```
struct record {     /* record is a tag */
    ...
} record;

typedef struct {    /* we can omit the tag */
    ...
} record;

typedef struct {
    ...
} r, a[10], *p;     /****/
```

the line r, a[10], *p is creating three different things

r is creating a struct of this type named r

a[10] is creating an array of ten of these structs called a

*p is declaring and reserving memory for an uninstantiated struct of this type

*OBSERVE: don't need to repeat struct record anymore*

*NOTE: we can combine the typedef and the definition of struct record at the same time:*

*FROM OUTSIDE THE COURSE: you can create structures (as C programmers usually do) in separate files and refer to them at the top using #define "fileName.h"*

```
#define NAMESIZE 20
#define IDSIZE 20

typedef struct {
    char first[NAMESIZE];
    char last[NAMESIZE];
} name;

typedef struct {
    char id[IDSIZE];
    name name;           /*OK, name is not a reserved word*/
    float score;
} record;

record comp2510[102];
```

*NOTE: we can just use the reference word record to create a single structure or an array of them because it is now a type*

## Assignment of structures

When you have two integer variables, you can store m = n

```
typedef struct {
    char id[10];
    int score;
} grade;
grade g1, g2;
strcpy(g1.id, "a66666666");
g1.score = 49;
g2 = g1;        /* assignment */
```

# November 1, 2018

## Pointers to void

A void pointer cannot be dereferenced

A void pointer is compatible with any other type of pointers to objects

```
int n = 123;
void *p = &n;   /* OK */
int *q &n;
p = q;          /* OK, compatible */
q = p;          /* OK, compatible */
*p = 456;       /* invalid, can't dereferencea void */
*(int *)p = 456 /* now is OK, changes to 456 */
q = p;
*q = 789;       /* changes to 789 */
```

*REMEMBER: void as in a void method that returns nothing*

You cannot dereference a pointer, because the destination is void, and there is no information about the destination type, thus it cannot be dereferenced

A void pointer is a pointer of no type (int, char, etc.) which it usually has

as such it can be assigned no value because it it typeless

you can however cast it to a type, and THEN assign a value to it

it can be pointed to any type of object (because it has no inherent type) but it can't point to functions

# Dynamic Memory

So far, the lifetime of objects are completely determined * if the variable in global, its lifetime is the duration of the block in which it is created.

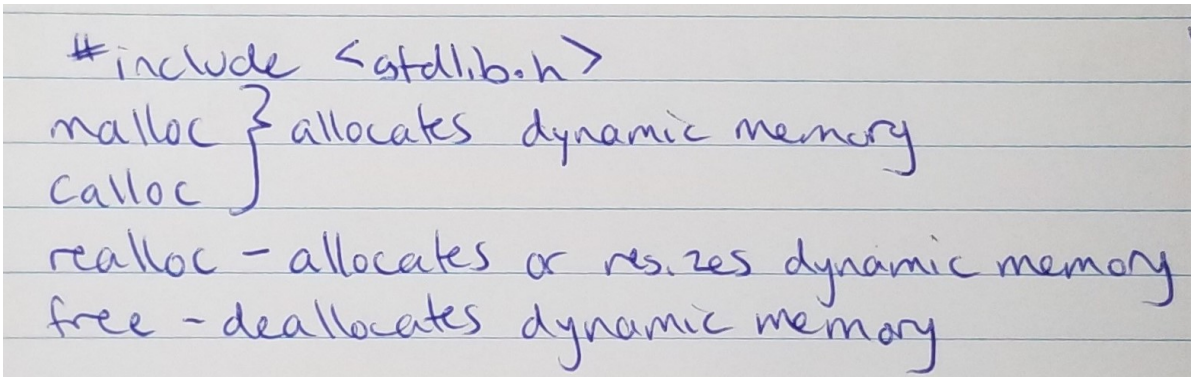
if the variable is global, it's lifetime is the duration of the program

if the variable is local, its lifetime is the duration of the block in which it is created

```c
int n = 10;          /* lifetime is duration of program */
int main(void) {
    int a = 1;       /* lifetime is duration of main */
    if (...) {
        int b = 2;   /* lifetime of b is duration of its statement */
        ...
    }                /* b destroyed here */
}                    /* a destroyed here */
```

Dynamic memory gives us more control over the lifetime of objects

you can allocate memory to store an integer in a function, and then access that integer outside a function




Header file is #include <stdlib.h>

malloc: allocated dynamic memory

calloc: allocates dynamic memory

realloc: allocates or resizes dynamic memory

free: deallocates dynamic memory

# malloc Function

**Example 1: Dynamic array of 100 ints**

malloc takes # of bytes and returns the starting address

```
int *p = malloc(100 * size(int));

if(p == 0) {        /* malloc failed */
    fprintf(stderr, "unable to allocate memory \n");
    /* additional error-hangling if needed such as 'exit(1);'*/
}
```

malloc fails with p == 0 because it means p is null

(100 * size(int)) denotes the number of bytes

whole first line returns starting address of allocated memory if successful

Assume malloc succeeds here from the previous example:

```
for (i = 0; i < 100; i++)       /* for example: */
    p[i] = 5*i;
    /* stores some numbers into dynamic array */
```

example is: store some numbers into the dynamic array

*REMINDER: a pointer is used to store an address*

Deallocate memory when we are done

```
free(p);
```

*NOTE: free(p) doesn't change the contents of p, rather it gets rid of the memory allocation, and whatever data was there remains there*

**Example 2: dynamic array of 100 records (struct record from last class)**

```
record *p = malloc (100 * sizeof(record));
if (p==0) {
    ...
}
strcpy(p[0].id, "a66666666");
...
```

What is the return type of malloc?

```
void *malloc(size_t);
/* malloc is of type void */

void free(void*);
/* free returns void */
```

## calloc & realloc Function

*IMPORTANT: calloc is similar to malloc except that it zeroes out the allocated memory; memory allocated by malloc contains random values*

*IMPORTANT: calloc takes TWO VALUES*

**Example:**

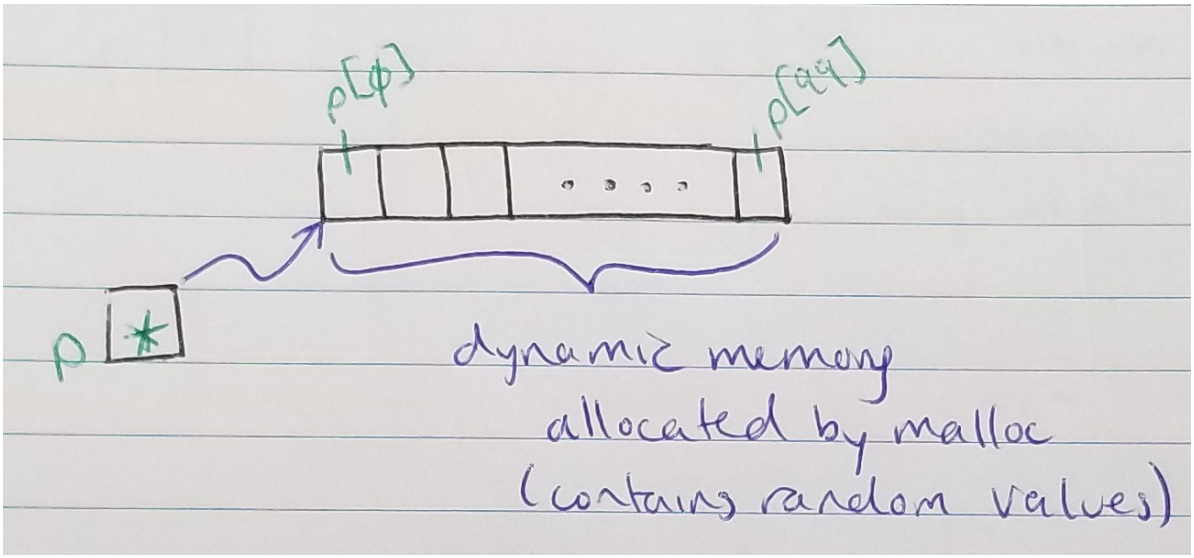
```
/* prototype of calloc */
void *calloc(size_t);

int *p = calloc(100, sizeof(int));

/* prototype of realloc */
void *realloc(void*, size_t);
```

```
int *p, *q;
if ((p = malloc(100 * sizeof(int))) == 0) {
    fprintf(stderr, "...");
    exit(1);
}
/* use the array here & find it is not enough */

/* resize the dynamic memory to 200 ints */
q = realloc(p, 200 * sizeof(int));
    /* realloc may fail, so we don't reassign to p */

if (q == 0) {
    /* realloc failed, continue to use the original p */
    ...
} else {
    p = q;
    /* use p which now points to 200 ints */
}

/* when done */
free(p);
```

# November 7, 2018

## Dynamic Memory

```
int *p = malloc(100 * sizeof (int))
```



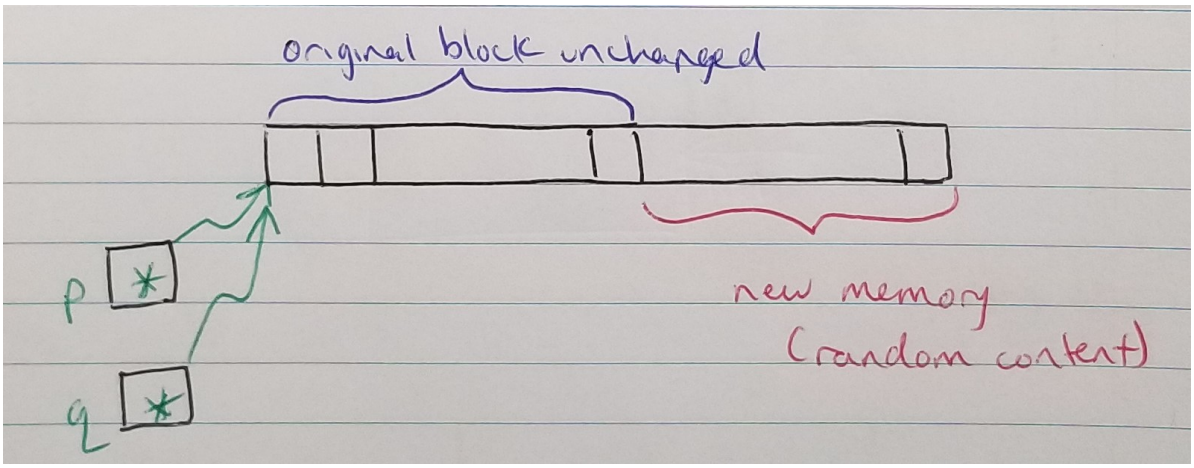

```
int *q realloc(p, 200 * sizeof (int));
```

*NOTICE: realloc takes two variables, the first one being the pointer to the memory it is trying to enlarge, the other being the standard allocation*

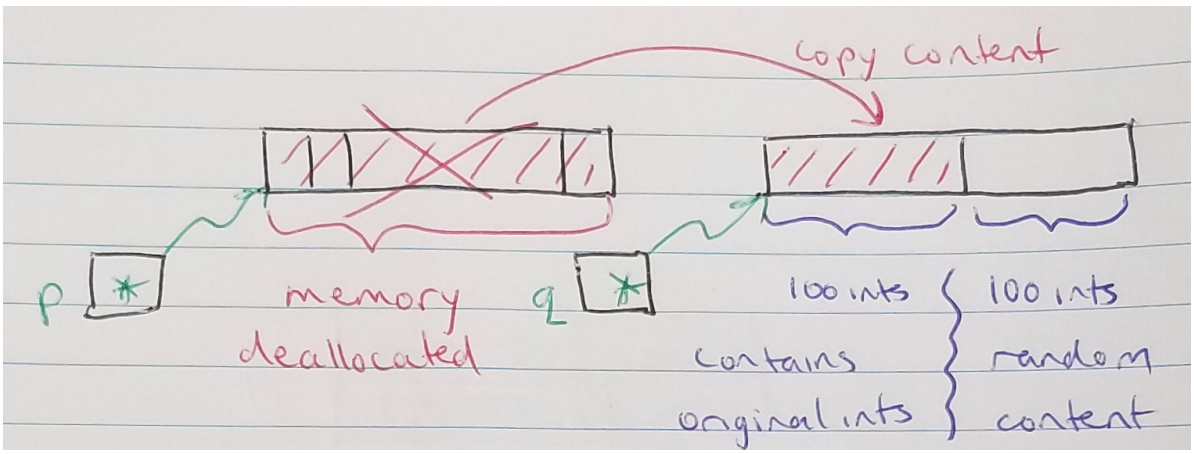
Assume this call to realloc succeeds

**2 possibilities**

1) there is enough memory just after the original block of dynamic memory

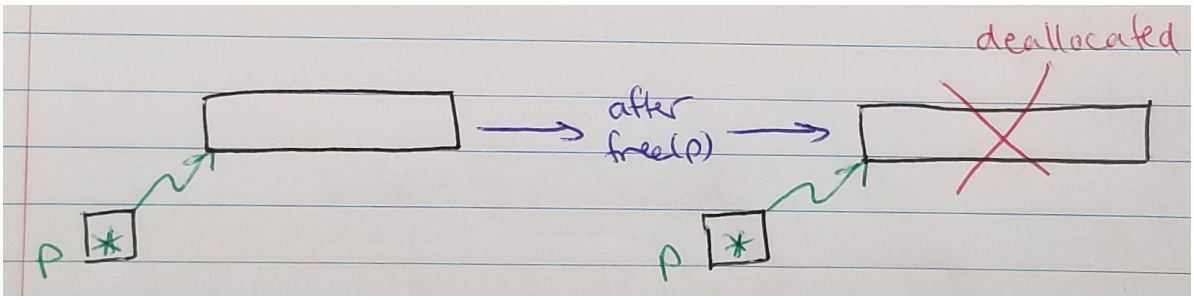



2) not enough memory next to the original




*NOTE: must denote *q in this process because you don't know which method will be used, and if you try to maintain the *p pointer you may be pointed to deallocated memory after the operation*

```
free(p);
```




p is deallocated, but still active so you could still call it by accident

therefore this is another way you must be careful in C

Special cases:

```
realloc(0, size) === malloc(size);
realloc(p, 0) === free(p);
```

## Storing Lines of Text: 3 versions

**1) Using a 2-Dimensional array**

```
#DEFINE NLINES 500
#DEFINE LINESIZE 128

char lines[NLINES][LINESIZE];

size_t i, j;

for(i = 0; i < NLINES; i++)
    if(!fgets(lines[i], LINESIZE, stdin))
        break;
for(j = 0; j < i; j++)
    printf("%s", lines [j]);
```

lines is an array of NLINES objects, each an array of LINESIZE chars

the first for loop determines how many entries there are in the array which i is left with the value of

then, j for loop will run as many times as i has gone up to

*REMIMDER: #DEFINE statements don't end in a semicolon!*

*REMINDER: fgets 'returns' the null character, which is zero, which is equivalent to false in C, and puts the line it has read into the buffer*

Disadvantages of this version is that we may waste memory because we need to choose large values for NLINES & NLINESIZE

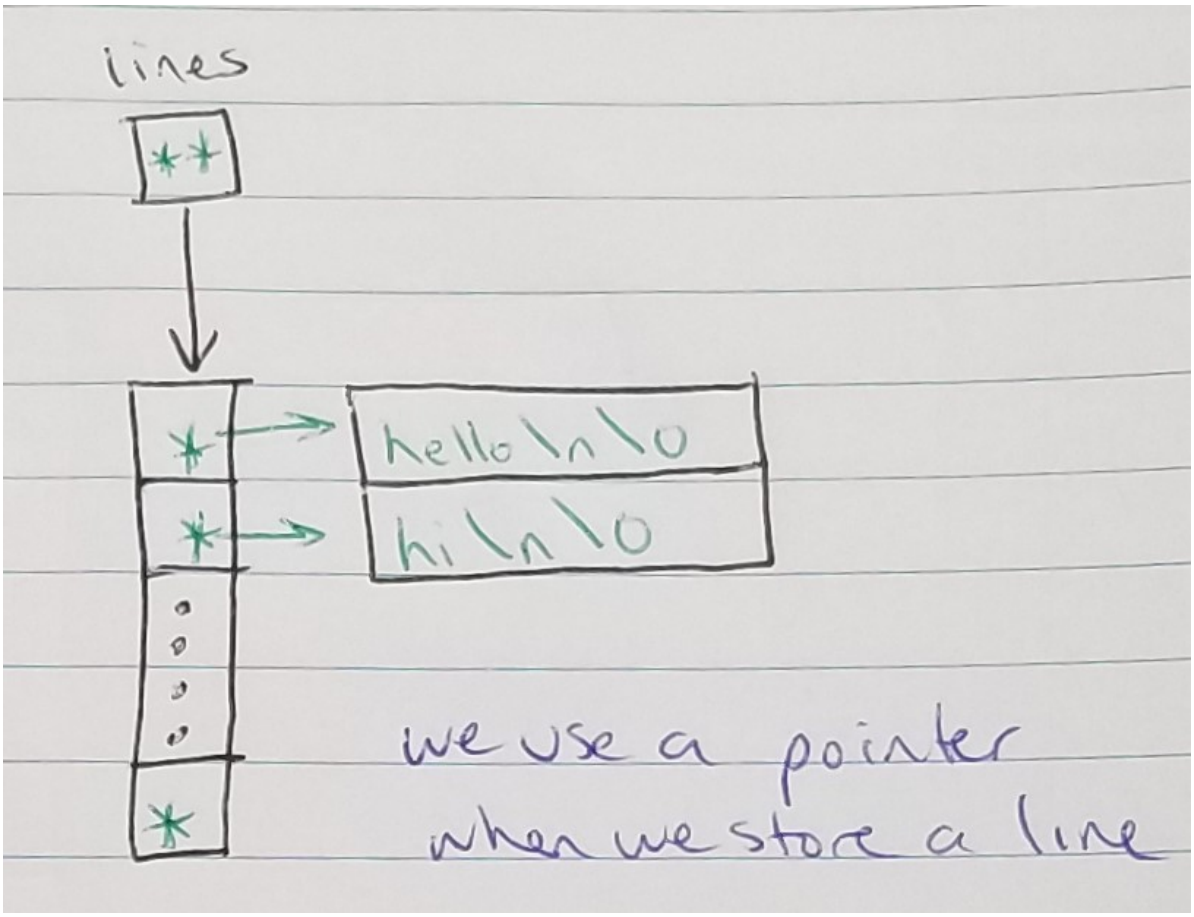
**2) The number of lines is fixed but each line is dynamically allocated**

```
#define NLINES 500
#define LINESIZE 1024

char *lines[NLINES];        /* array of 500 line pointers */
char *buffer[LINESIZE];
size_t i, j;

for (i = 0; i < NLINES; i++) {
    if(!fgets(buffer, LINESIZE, stdin))
        break;
    lines[i] = malloc(strlen(buffer) + 1);  /* 1 is for null character */
    if (lines[i] == 0) {
        fprintf(stderr, "malloc failed \n");
        break;
    }
    strcpy(lines[i], buffer);
}

for (j = 0; j < i; j++);
    printf("%s", lines[i]);

/* deallocate memory */
for (j = 0; j < i; j++)
    free(lines[j]);
```

*REMINDER: strlen does not count the null character*

Limitation is we can read at most 500 lines

**3) Make lines in 2 versions of a dynamic array**



```
#define LINESIZE 1024
#define BLOCK 100

char **lines;
size_t nalloc, nused;
char buffer[LINESIZE];

nalloc = nused = 0;
lines = 0;

while (fgets(buffer, LINESIZE, stdin)) {
        /* if we have used up the pointers */
    if(nalloc == nused) {
        /* we need to allocate more */
        char **tmp = realloc(lines, (nalloc + BLOCK) * sizeof(char *));
        if (tmp == 0) {     /* realloc failed */
            fprintf(stderr, "realloc failed \n");
            break;
        }
        lines = tmp;
        nalloc += BLOCK;
    }

    /* there are unused pointers */
    lines[nused] = malloc(strlen(buffer) + 1);
    if lines[nused] == 0) {
        fprintf(stderr, "malloc failed\n");
        break;
    }
    strcpy(lines[nused++], buffer);
        /* nused++; */
        /* could have incremented on second line */
}

for (i = 0; i < nused; i++)
    printf("%s", lines[i]);

/* deallocate memory when we are done */
for (i = 0; i < nused; i++)
    free(lines[i]);
free(lines);
```

nalloc is the number of pointers allocated

nused is the number of those pointers used

while statement means as long as we can read a line

*NOTE: all pointers take up the same amount of memory, regardless of the type they point to*

DO NOT DO:

```
p = realloc(p, ...);
```

i.e. do not directly assign back to p when we realloc p

if you do this and realloc fails, p will contain 0 (the null pointer), and we will have lost the address of the original block of memory

# qsort

used to sort an array

**Example:**

sort an array of 100 ints in ascending order

```
#include <stdlib.h>

int a[100];
    /* assume we've stored 100 ints in a */
qsort(a, 100, sizeof(a[0]), cmp);
```

first parameter of qsort  is the array name

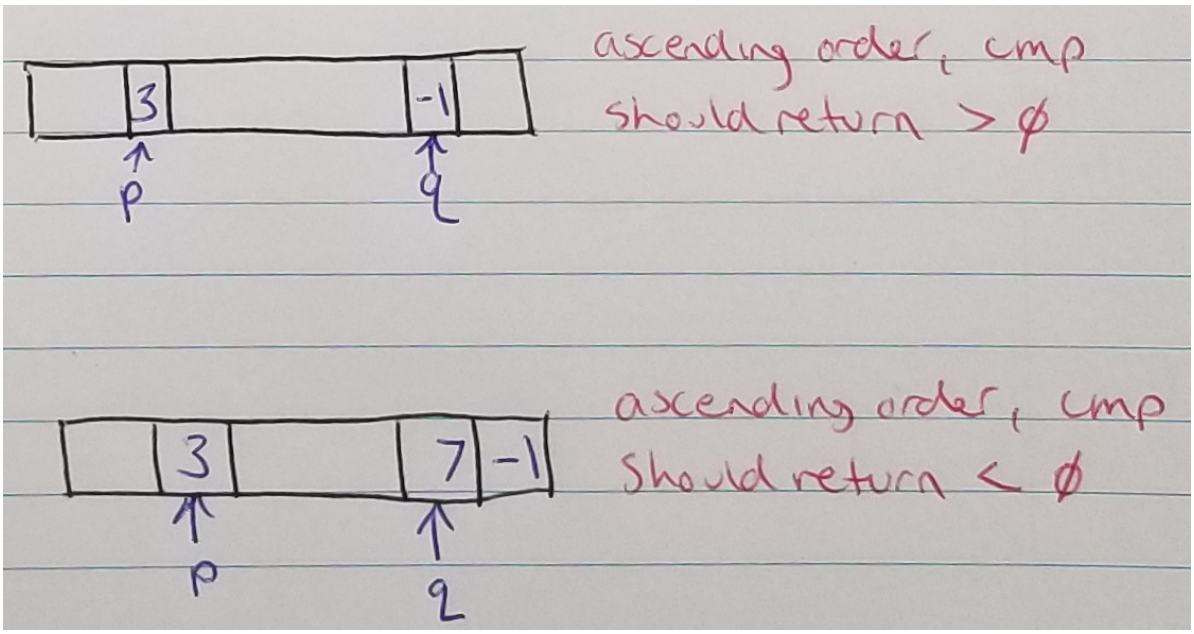
second parameter of qsort is the number of elements

third parameter of qsort is the size of each element

fourth parameter of qsort  is the comparison function, used to specify the sorting order

cmp is used to compare 2 elements in the array

cmp is actually a function which is called by qsort, passing in *the addresses of the* two elements it wants to compare




cmp(p, q) should return a negative integer if *p should go before *q after sorting, positive integer if *p should go after * q after sorting, and 0 if it does not matter which goes first after sorting

```
int cmp( const void *p, const void * q) {
    const int *pp = p;
    const int *qq = q;
    return *pp - *qq;   /* ascending order */
}
```

*OBSERVE: cmp should return a negative value if * p should go before * q, but a positive value if * q should go before *  p*

# November 8, 2018

**Example: sorting an array of strings**

```
char *a[] = {"hello", "goodbye", "world", ...};
    /* array of pointers to char */

/*sort a in ascending order */
qsort(a, (sizeof(a) / sizeof(a[0]), cmp);
      /* division to determine number of elements in array */
int cmp (const void *p, const void *q) {    /* array element is a constant */
    char * const *pp = p;   /* double p points to a constant pointer, meaning
the array                                    element is a constant pointer */
    char * const *qq = q;   /* where the const need to go */
    return strcmp(*pp, *qq);
}
```

**Example: sorting grades**

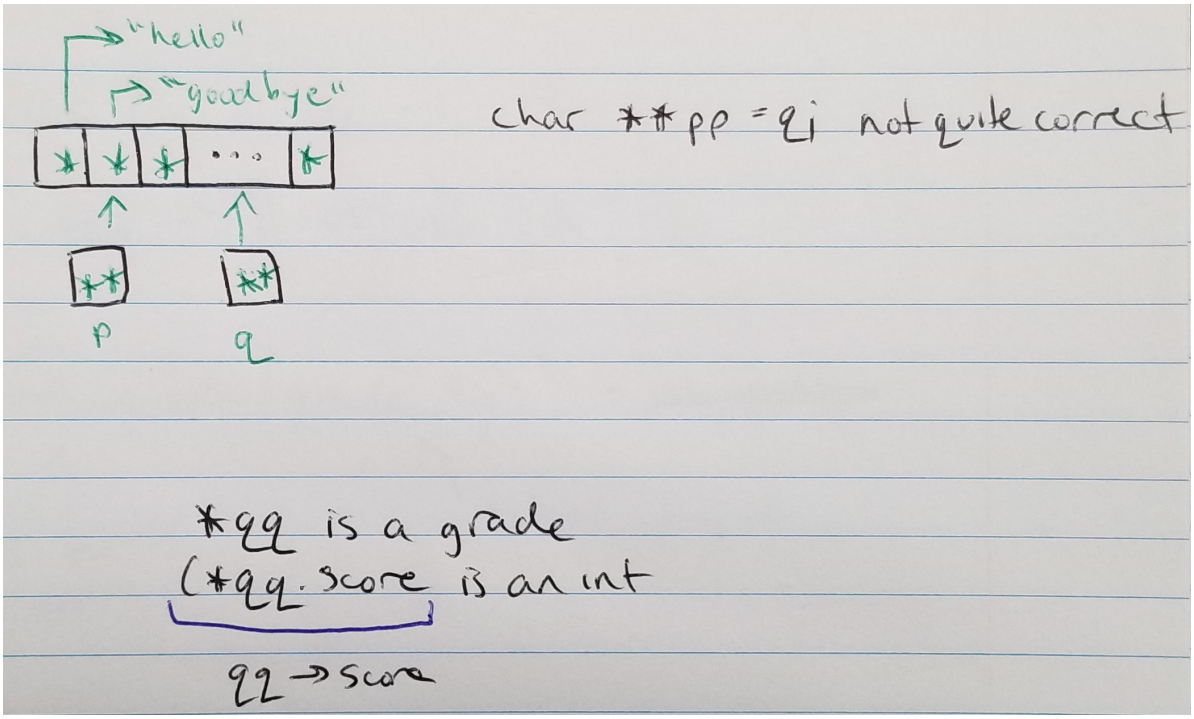
```
typedef struct {
    char id[10];
    int score;
} grade;

grade a[100];
/* assume we've stored 100 grades in a */
```

Sort in descending order of scores if 2 or more grades have the same score, they are sorted in ascending order of their IDs

*NOTE: function can be named anything other than cmp*

```
/* EXAM QUESTION!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! */

int cmp(const void *p, const void *q) {
    const grade *pp = p;
    const grade *qq = q;
    int n;
    if((n = qq -> score - pp -> score) != 0)
                            /* descending order of score */
        return n;
    return strcmp(pp -> id, qq -> id);
                            /* ascending order of id */
}
```

*IMPORTANT: in a compare, if it is (p, q), ascending is p, then q, descending is q, then p*




```c
#include <stdio.h>
#include <stdlib.h>

typdef struct {
    char id[10];
    int score;
} grade;

int cmp(const void *p, const void *q) {
    const grade *pp = p;
    const grade *qq = q;
    int n;

    if ((n = qq -> score - pp -> score) !=0)
        return n;

    return strcmp(pp -> id, qq -> id);
}

int main (void) {

    grade a[100];
    size_t i, j;

    for (i = 0; i < 100; i++);
        if (scanf("%9s %d", a[i].id, &a[i].score)) != 2)
            break;
        qsort(a, i, sizeof(grade), cmp);
        for(j = 0; j < i; j++)
            printf("%s: %d\n", a[j].id, a[j].score);
    return 0;
}
```

*NOTE: need a pointer to a grade because you are not going to store something in it*

# November 14, 2018

What is the prototype of qsort?

*NOTE: qsort doesn't return anything if type is void*

```
void qsort(void *, size_t, size_t, ???)
```

void * is starting position of array

first size_t is the number of elements,

second size_t is the size of the elements

fourth element is the sort function

# Function Pointers

A function pointer is a variable that stores the address of a function

### 3 need to be able to

1) declare a function pointer

2) take the address of a function

3) call the function specified by a function pointer

# Address of a function

If f is a function, we can use &f for the address of f

*IMPORTANT: We can also use f as the address of the function*

```
void f (int);   /* we can use &f or f as the address */
```

# Calling a function via a function pointer

*REMINDER: a pointer points to a thing if the pointer contains the address of the thing*

Suppose p points to the function f with prototype void f(int)

i.e. p = f or p = &f

We can call f via p passing in 2 as the argument follows:

```
(*p)2;      /* or simply */     p(2);

q = &g;     /* where */         int g(int, int);
    /* q is a function pointer */

printf("%d\n", q(1,2));
```

*REMINDER: brackets have very high precedence*

# Declaring a function pointer

**Examples 1:**

```
void f(int);
void(*p)(int)
    /* *p is a pointer to function that takes an int and returns nothing */
```

**Example 2:**

```
int g(int, int);
int (*q)(int, int) = g;     /* (*q)(1,2) or simply q(1,2) */
```

**Example 3:**

qsort prototype -- it uses a function pointer

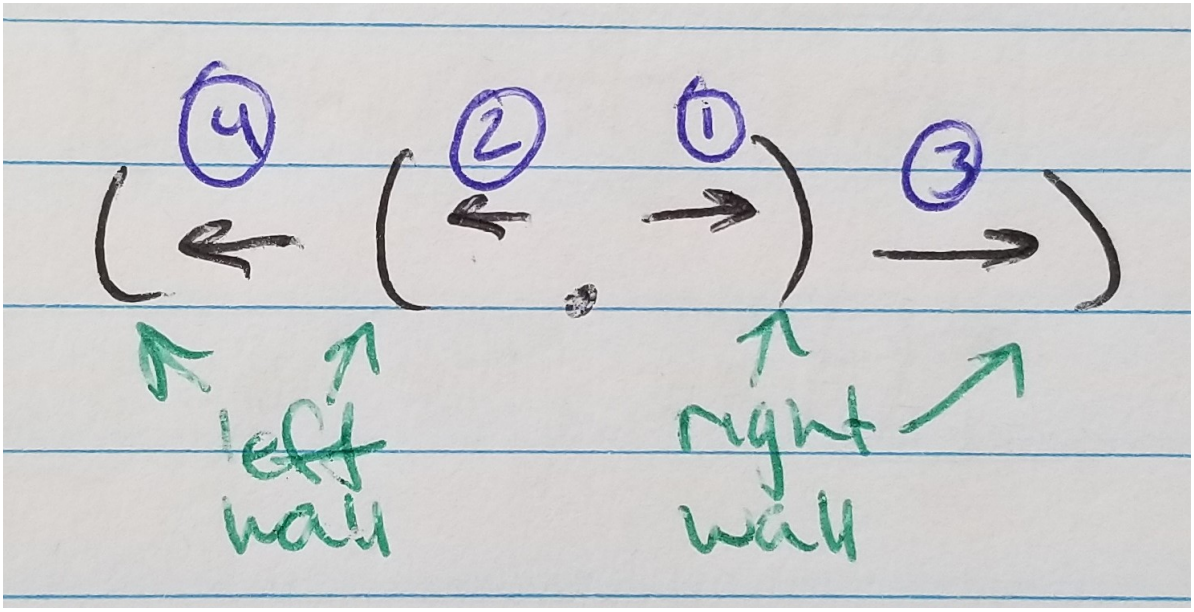
```
qsort(a, 100, sizeof(a[0]), cmp);

void qsort(void *, size_t, size_t, int(*)(const void *, const void *));
```

# Right -Left Rule

Tells us how to read a declaration, therefore by reading it backwards we will be able to write the declaration

**Example 1**

```
void (*p)(int, int);
    /* done with the innermost pair of walls and jump out */
```




*IMPORTANT: when doing the wall things, start with the name*

(2) p is a pointer to (3) a function that takes 2 ints and (4) returns nothing

**Example 2**

```
const int * const p
```

p is a constant pointer to integer constant

**Example 3**

```
int * a[3];
```

a is (1) an array of 3 (2) pointers to int)

*NOTICE: you include the array of THREE before turning left*

**Example 4**

```
int (*a)[3];
```

a is (2) a pointer to (3) array of 3 (4) ints

*NOTICE: you bounce off the right bracket first, then the left after seeing the pointer*

What is the difference between a pointer to an int and a pointer to array of 3 ints? A pointer to an int can already point to an array of ints

```
a[0] is an int but q[0] is an array of 3 ints --> q [0][0] is an int
```

**Example 5**

```
void f(ints);
void g(int);
void(*a[2])(int) = {f,g};
```

a is an array of 2 pointers to functions that take an int and return nothing

*NOTICE: the way you have to interpret take in an int and return nothing*

void (*a)[2] (int) illegal - can't have array of functions

**Example 6**

```
int a [2][3];
```

a is (1) an array of 2 objects each an array of 3 (2) ints

*NOTICE: you go right through the first set of square brackets through the second*

Example of application function pointer

**Version 1**

```
/* get 2 ints a and b from user */
int choice = get_valid_choice();
switch (choice) {
    case 0:
        printf("%d\n", a + b);
        break;
```

```
    case 1:
        printf("%d\n", a - b);
        break;
    case 2:
        printf("%d\n", a * b);
        break;
    case 3:
        printf("%d\n", a / b);  /* divide by zero */
        break;
}
```

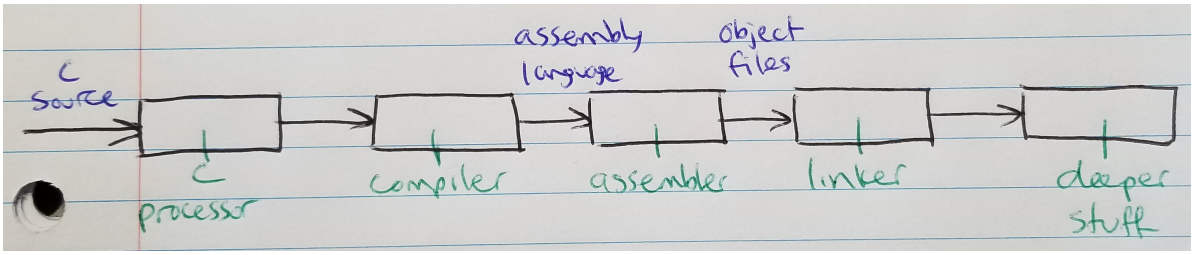
**Version 2**

```
int (*ops[])(int,int) = {add, subtract, multiply, divide};
int choice = get_valid_choice();
printf("%d\n", (ops[chioice])(a,b));
    /* index into array of function pointers */
```

*QUESTION: where does it define what the behaviours of add/subtract/multiply/divide*

# The C Preprocessor

Stages in compiling a C program & generating an executable:




-------(C source file)------> C preprocessor ------(preprocessed output)----->

Compiler ------(assembly language)-----> Assembler ------(object files)----->

Linker ------(executable)----->

The preprocessor reads in the context of the header file & puts it at the location of the # statements

C processor handles preprocessor directives. Preprocessor directives are found in the source code.

## Some preprocessor directives:

**1) #include: 2 Versions:**

i) *#include <stdio.h>*

header file

preprocessor looks for this in a system directory

ii) *#include "lab10.h"*

preprocessor defaults to look for this in the current directory

*#include "headers/lab10.h"*

essentially, the preprocessor reads in the content of the header file and puts it at the location of the #include statement

**2) #define --> used to define macros or symbolic names**

i) *#define LINESIZE 1024*

char line[LINESIZE] ----> char line[1024]

process called macro expansion ( performed by text replacement)

"LINESIZE is 1024" --> not replaced because it is in a string

ii) *#define LINESIZE 1024*

char line[LINESIZE]; --> char line[1024;]; --> ERROR!!!

ii) using macros as functions:

```
#define LINESIZE 1024;      /* does not compile */
char line[LINESIZE;];       /* does not compile after macroexpansion */


#define LINESIZE 1024
char line[LINESIZE];        /* preprocessor changes this to line[1024] */

#define SQUARE(X) X*X
SQUARE(5); -----> 5 * 5;                    /* okay */
SQUARE(4 + 1); ------> 4 + 1 * 4 + 1;       /* a problem */

#define SQUARE2(X) (X)*(X)
SQUARE(4 + 1); ------> (4 + 1) * (4 + 1);    /* okay */
25/SQUARE2(5); ------> 25/(5)*(5);          /* a problem */

#define SQUARE3(X) ((X)*(X))
25/SQUARE3(5); ------> 25/((5)*(5));        /* okay */
        /* use many brackets when writing macros */

int n = 1;
SQUARE3(n++); --> ((n++)*(n++));
    /* n modified more than once, value unspecified */
    /* when using macros, do not use incrementors */
```

*NOTE: LINESIZE is a macro, the preprocessor performs macro expansion (performed via text replacement). Whenever the preprocessor sees LINESIZE it replaces the text with 1024. The macro isn't replaced if it is found in strings. It is not a variable name because the compiler doesn't even see it*

**3) Conditional Conclusion/Compilation**

#if / #elif / #else / #endif

#ifdef/ #endif

ifndef / #endif

**Example 1**

```
gcc -ansi -W -Wall -pedantic -Werror -DDEBUG
```

The extra -D here means define

if -DDEBUG isn't specified in bash, the block of code between #ifdef and #endif will never be executed and it is gone from the executable

```
#ifdef DEBUG
    /* if DEBUG is defined in command line (-DDEBUG) */

fprintf(stderr, "...");
    /* then do this */

#endif
```

**Example 2**

```
gcc -ansi -W -Wall - pedantic -Werror -DLINEIZE=512
```

```
#ifndef LINESIZE
    /* if LINESIZE is not defined */

    #define LINESIZE 1024
        /* then define LINESIZE as 1024 */

#endif
```

**Example 3 - test.c**

```
#ifndef LINESIZE
    #define LINESIZE 1024
#endif
#include <stdio.h>
#define SQUARE(X) (X)*(X)

int main(void) {
#if 0
    char line[LINESIZE];
    SQUARE(1 + 2);
#endif

#ifdef DEBUG==1
    fprintf(stderr, "debugging\n");
#elif DEBUG==2
    fprintf(stderr, "debugging level 2\n");
#else
    fprintf(stderr, "no debugging\n");
#endif
    return 1;
```

*QUESTION: how does #if 0 work?*

```
gcc -E test.c               # show text replacement
gcc -DDEBUG -E test.c       # define debug
gcc -DDEBUG=1 -E test.c
gcc -DDEBUG=2 -E test.c
gcc -DDEBUG=99 -E test.c
gcc -DLINESIZE=512 -E test.c
```

*NOTE: when you enter the -E command, you will see all of the macros and includes included in the file*

# November 15, 2018

#ifndef is used to implement include guards

We don't want a file to include the same header file name more than once

include guards are used to prevent this from happening

in the header file, add these lines:

```
#ifndef HEADER_H
    /* choose appropriate macro names */
    /* by convention should be related to file name */

    #define HEADER_H
        /* content of header file goes here */

#endif
```

*NOTE: '.' dots are not allowed, so dots are replaced with underscore '_'*

**Example - Clearing a Screen**

```
#define DOS 1
#define UNIX 2

int main(void) {
    #if OS == DOS
        system("cls");
    #elif OS == UNIX
        system("clear");
    #else
        /* print \n 1000 times */
}
```

Compile above using the following bash line:

```
gcc -ansi -W -Wall -Werror -pedantic -DOS=DOS
        # -D defines
```

clears the screen using specific OS commands or prints new line 1000 times

```
#define CHECK(x) printf("%s....%s\n", (X)? "passed":"FAILED"\
    #X)
```

*NOTE: if you include a backslash at the end of a line you can continue the macro on the subsequent line*

# November 21, 2018

## The static keyword

Static has 2 meanings

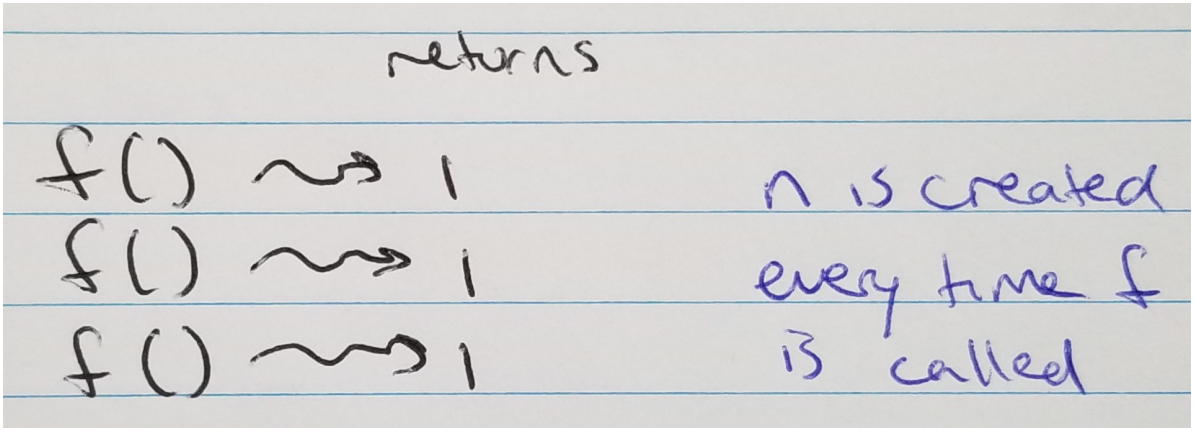
when applied to a local variable, it extends its lifetime to the duration of the program

when applied to an external (global) variable or a function, it specifies static (also called internal) linkage.

**Example**

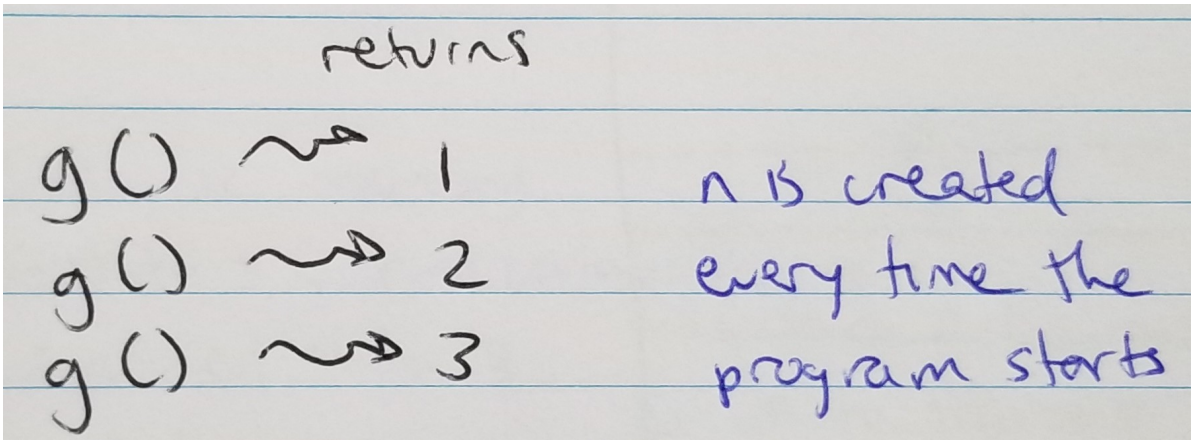
```
int f (void) {
    int n = 1;
    return n++;
}
```

no matter how many times you call f() it will always return 1.  There is always a new 'n' being returned




```
int g(void) {
    static int n = 1;
    return n++;
}
```

each successive call of g() will return its old value plus any modifications




NOTE: an uninitialized static variable contains 0

REMINDER: an external variable defaults to 0

an uninitialized non-static local variable has a random value

# Linkage:

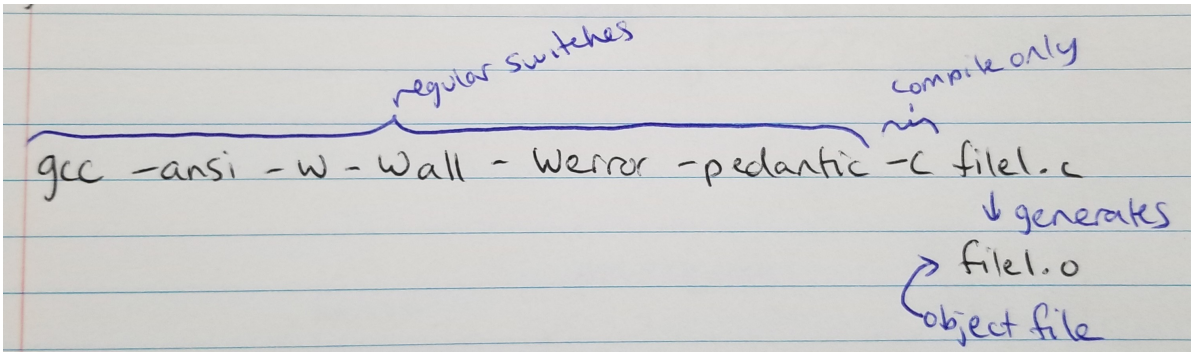
static (internal) vs external linkage

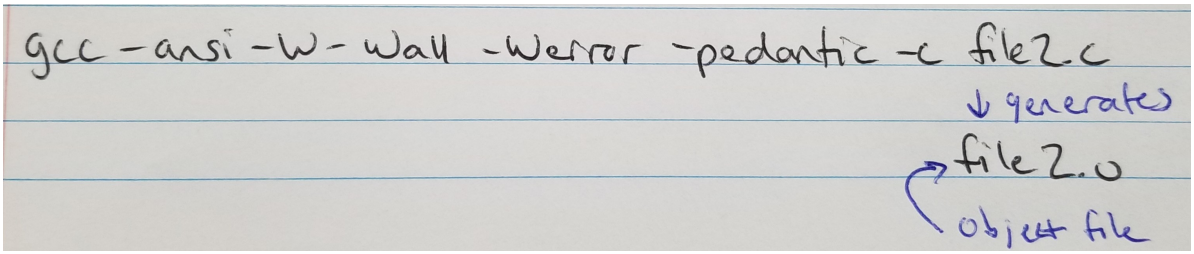
we can break up a large program into multiple C source files

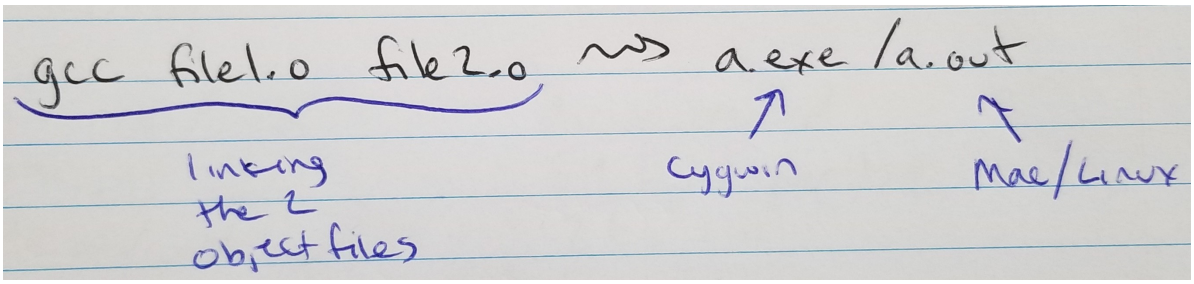
we can compile each c file individually and then link them together

**Example: Linking Two C Files Together**

2 C files: file1.c and file1.c




*NOTE: The switch -c means compile only. The above bash line generate object files name file1.o and file2.o*

*REMINDER: elements after gcc are called 'switches'*

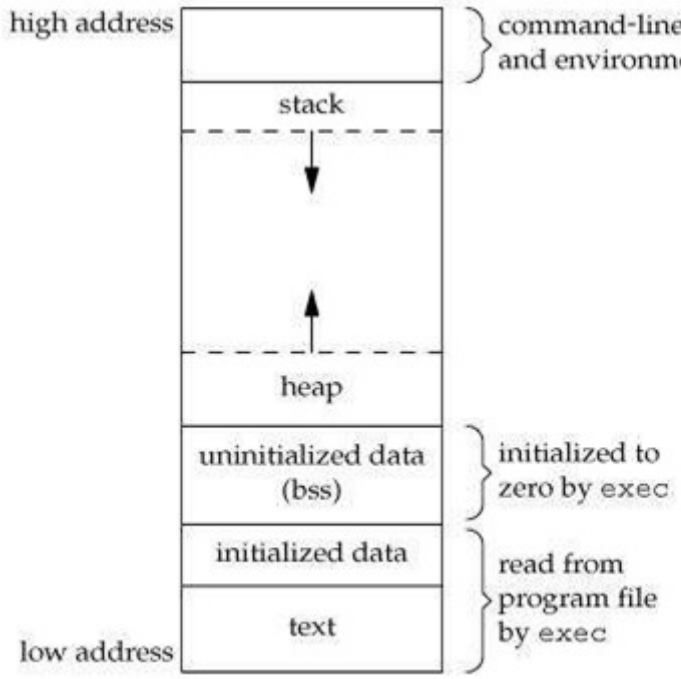
*NOTE: The third gcc line LINKS the 2 objects*

If a function or external variable has external linkage, it can be linked from another file.

If the linkage is static, then it cannot be linked from another file

*QUESTION: what does it mean for something to have 'external linkage'*

# Layout of a program in memory




The absolute lowest part near zero (not shown in this diagram) is reserved so that zero is never called, since zero is the null pointer and needs to be protected

*text* = machine code of the program

*initialized data* = static and external variables

```
int a[] = {3, 2, 7, 6, 8};
static int n = 1;
```

*bss* = uninitialized static and external variables (contains 0s)

*REMINDER: uninitialized static and external variables contain 0s and that uninitialized non-static variables contain random values*

```
static int m;
```

heap = dynamic memory

stack = (non-static) local variables, function arguments, etc

```
int a;
```

# Multiple C source files

In general, each .c file (except the one containing main) should have a corresponding .h file. The .h file contains the prototypes of functions "exported" by the .c file

Always use include guards in header file

Functions in a .c file that are not exported should be declared static

Never compile header files; they are intended to be included with #include

If you need to refer to something in another file, include the corresponding header file

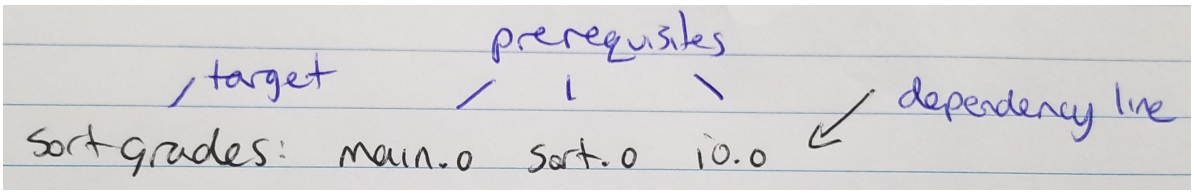
Compile each.c file using the -c switch in gcc to generate a .o file (object file)

Link all of the object files with:

```
gcc * .o
```

# Makefile

Makefile is used to specify dependencies




## The Dependency Line

```
sortgrades: main.o sort.o io.o
```

*NOTE: This is called a 'dependency line'. 'sortgrades' is called the target, 'main.o sort.o io.o' are the prerequisites*

## The Command Line

```
    gcc main.o sort.o io.o -o sortgrades
```

*NOTE: This is called a command line*

*NOTE: command lines must be indented under the dependency line*

*NOTE: indentation must be tab, cannot be spaces*

```
main.o: main.c grade.h io.h sort.h
    gcc iansi-c -w -Wall -Werror -pedantic -c main.c    # command line
            # must be a tab, cannot be spaces

sort.o: sort.c grade.h
    gcc -ansi -w -Wall -Werror -pedantic -c io.c

io.o: io.c grade.h
    gcc -ansi -w -Wall -Werror -pedantic -c io.c
            # -ansi to -pedantic can be replaced with $(CFLAGS)

$ make CFLAGS = -ansi -w -Wall -Werror -pedantic
```

# The Makefile

$@ = shorthand for the target

$^ = right side (all prerequisites)

$< = first prerequisite

```
cc = gcc
CFLAGS = -ansi -W -Wall -Werror -pedantic

sortgrades: main.o sort.o io.o
    $(CC) $^ -o $@

main.o: main.c grade.h sort.h io.h
    $(CC) $(CFLAGS) -c $<

sort.o: sort.c grade.h
    $(CC) $(CFLAGS) -c $<

io.o: io.c grade.h
    $(CC) $(CFLAGS) -c $<

clean:          # removes all .o file
    rm -f *.o
```

*IMPORTANT: Make is a program used to build programs. Other examples of dependency program are Maven and Ant. Make was the first dependency program and is famous for that reason. It is provided on a typical system although you may not have it installed in cygwin*

# Pointer Arithmetic

Example

```
char *p = "hello";
```

Assume that p has the value: 3276840

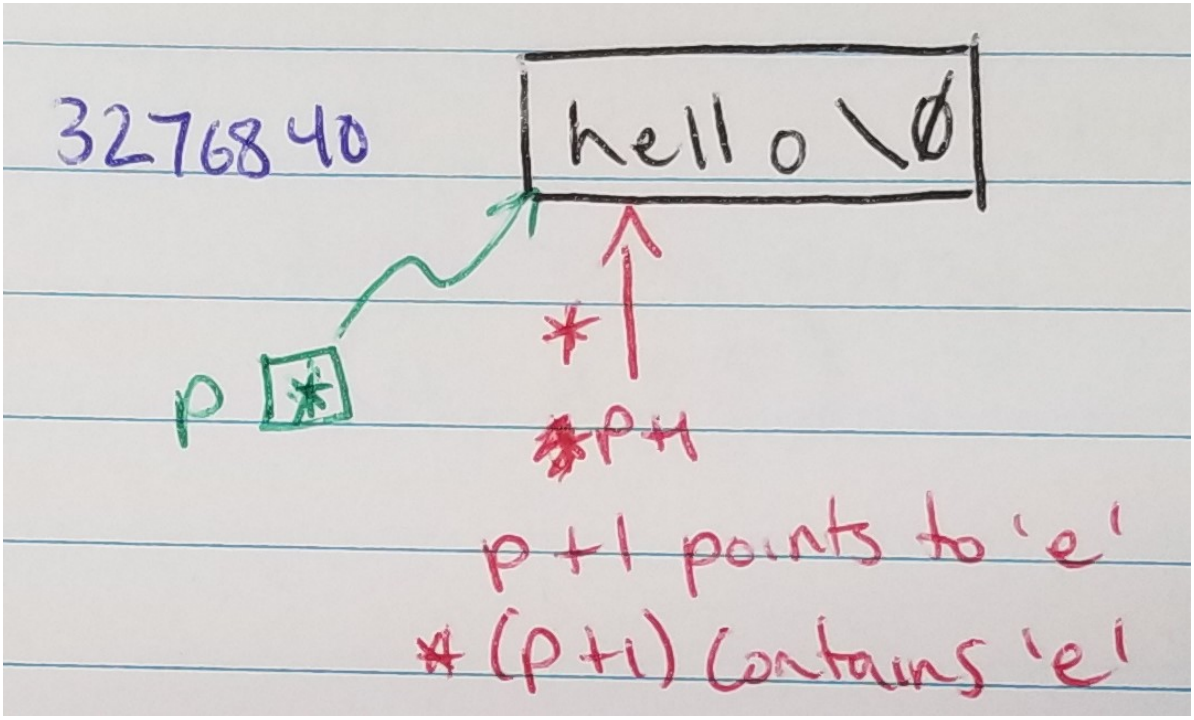
What is the value of p +1??

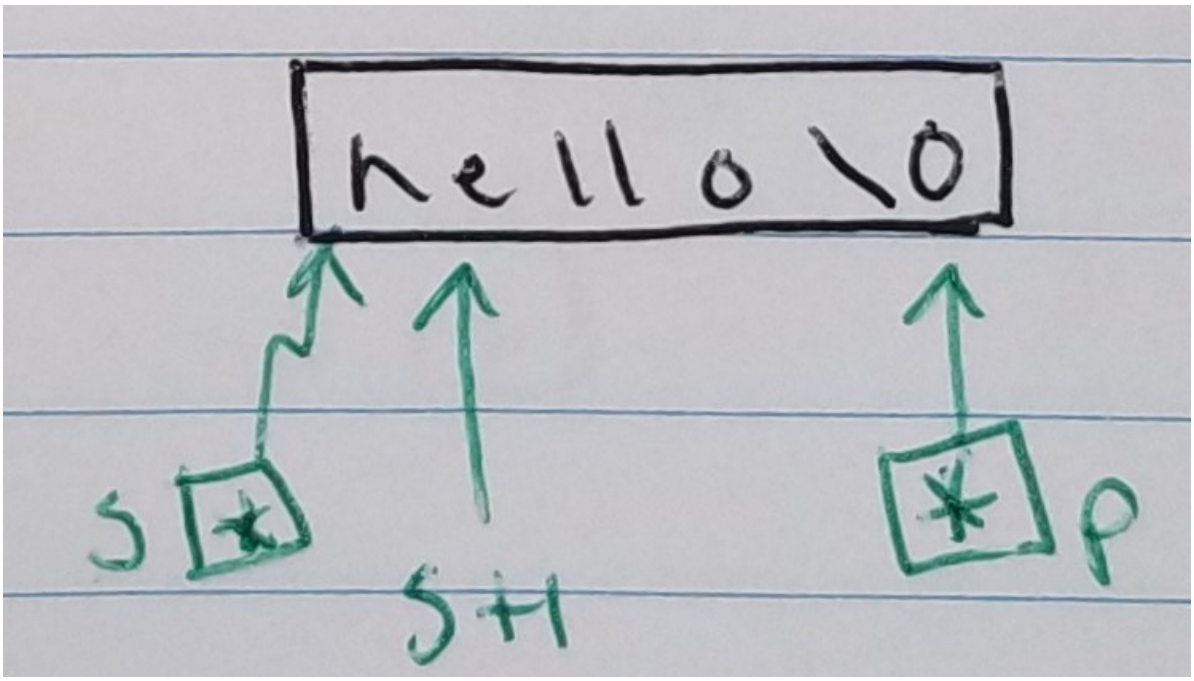
answer: 3276841

What is the value of p+2??

answer 3276842

What is p + 1 pointing to




p is NOT a pointer, but is IS the address of e

as in it is not a variable

p+1 points to 'e'

Once we know p + 1, we can define p + n for any non-negative integer n

We can also define p - n, p++, p --, ++p, --p, p += n, p-=n

P + 2 == (P + 1) +1

We can also define p == q, p != q, p > q, p >= q, p < q, p <= q (for 2 pointers p + q)

## Standard Idiom to Process a String (Pointer Version)

```
/* s is a string */

(const) char *
/* optional, depending on whether or not we are altering the string*/
for (p = s; *p != '\0'; p++)
        /* procdss *p */
```

**Example 1: length of a string**

```
size_t str_length(const char *s) {
    const char *p;
    for (p = s; *p != '\0'; p++)
        return p - s;
                /* address of null character minus address of string */
}
```

use a temporary pointer to loop through characters of s

**Example 2: changing a string to all uppercase**

```
void str_uppercase(char *s) {
    char *p;
    for (p = s; *p != '\0'; p++)
        *p = touper(*p);
}
```

p is not really necessary in this case since s is already local to the function

Simpler version:

```
void str_uppercase(char *s) {
    for(; *s != '\0'; s++)
        *s = toupper(*s);
}

/* option to declare something is declined */

/* even simpler */

void str_uppercase(char *s) {
    for(; *s; s++)
        *s = toupper(*s);
}

/* says while *s is true, as in not pointing to zero */

void str_uppercase(char *s) {
    for(; *s;)
        *s = toupper(*s++);
}

void str_uppercase(char *s) {
    while((*s = toupper(*s++)));
}

/* a = 10 is also returning 10 */
```

**Example 3: looking for a character in a string**

strchr is a function in the standard c library which returns a pointer

```
char * str_find(const char *s, int c) {
    const char *p;
    for(p = s; *p != '\0'; p++) {
        if (*p == c)
            return (char *) p;
                /* correcting mismatch between const char* and char* */
    return 0;
    }
}
/* can't pass a constant into a non constant parameter of a function */
/* casting from a const char * to a char */

/* alternately */

char * str_find(const char *s, int c) {
    char *p = s;
    for(; *p != '\0'; p++) {
        if (*p == c)
            return p;
                /* correcting mismatch between const char* and char* */
    return 0;
    }
}
```

the function returns a char * instead of a const char * so that we use the returned pointer to change the found character

**Example 4: function to copy a string**

**Version 1:**

```
void str_copy(char * dest, const char * src) {
    char *d;
    const char *s;
    for (s = src, d = dest; *s != '\0'; s++, d++)
        *d = *s
    *d = *s;
}
```

**Version 2:**

```
void str_copy(char * dest, const char * src) {
    for (; *src != '\0'; src++, dest++)
        *dest = *src
    *dest = *src;
}
```

**Version 3:**

```
void str_copy(char * dest, const char * src) {
    while(*src != '\0')
        *dest++ = *src++;       /* *p++ ==*(p++) due to precedence table */
    *dest = *src;
}
```

**Version 4:**

```
void str_copy(char * dest, const char * src) {
    while((*dest++ = *src++) != '\0')
        ;
}
```

**Version 5:**

```
void str_copy(char * dest, const char * src) {
    while((*dest++ = *src++));  /* double brackets prevent compiler warning */

}
```

# November 28, 2018

## Pointer arithmetic (anything aside from characters)

```
int a[] = {3, 2, 7, 5, 8}
int *p = a;     /* same as in *p = &a[0]; */
```

Assume the value of p is 3276800

What is the value of p + 1?

It turns out that it is not 3276801 but 3276804 9 (for 32-bit ints)

this is the address of a[1]

p+1 points to a[1]

p+2 points to a[2]...

using this, we can define p+n where n is some integer, p++, ++p, p--, --p, p-q where both are pointers

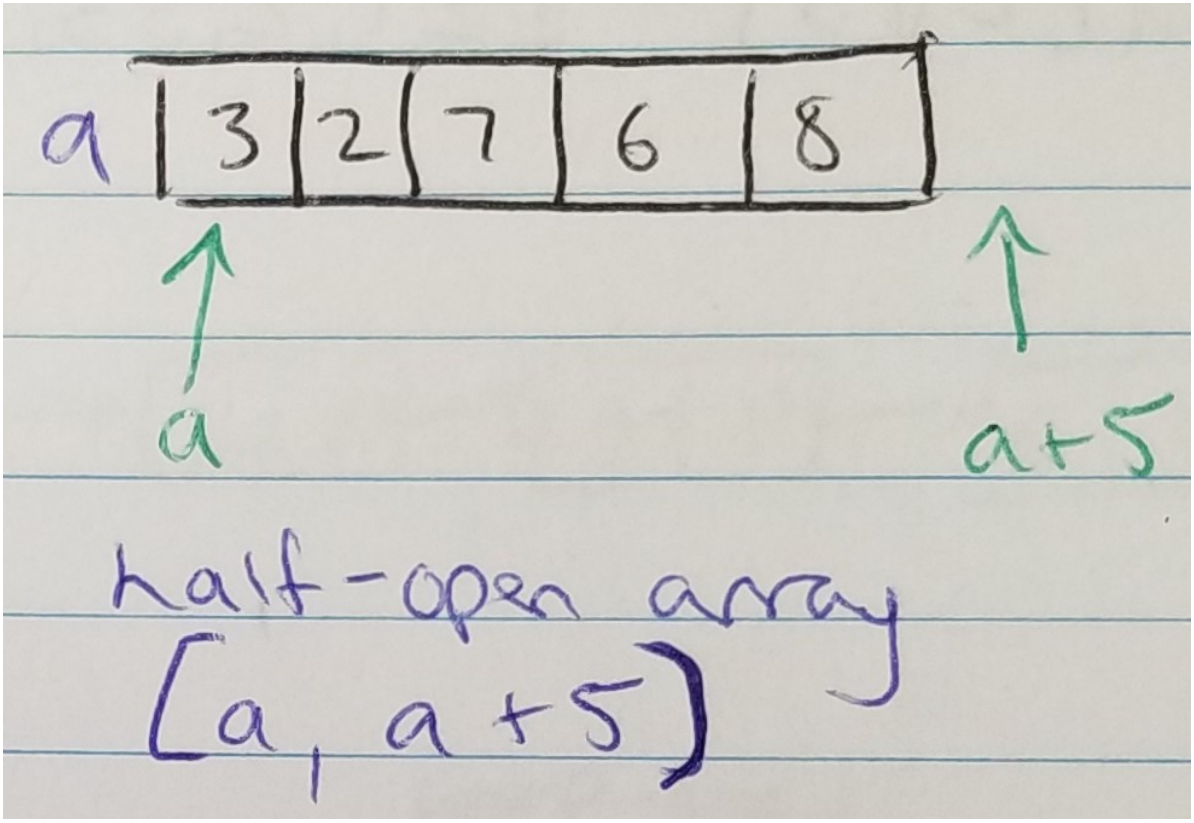
## Standard Idiom to process an array (Pointer Version)

```
T a[N]; /* T is some type, N is some positive integer */

(const) T * p;
    for(p = a; p < a + N; p++)
        /* process *p */
```

### Example 1 - Summing an integer array

```
int arr_sum(const int * a, size_t n) {
    const int * p;
    int sum = 0;
    for (p = a; p < a + n; p++)
        sum += *p;
    return sum;
}
```

### Example 2 - maximum value in a non-empty integer array




```
/* precondition; n >= 1 */
int arr_max(const int *a, size_t n) {
    const int *p;
    int max = *a;
    for (p = a; p < a + n; p++) {
        if (*p > max)
            max = *p;
    }
    return max;
}
```

### Example 3 - looking for an integer in an integer array

```
int * arr_find(const int * a, size_t n, int x) {
    const int * p;
    for (p = a; p < a + n; p++) {
        if (*p == x)
            return (int *)p;
    }
    return 0;
}
```

**Example 4 - tripling the integers in an integer array**

```
void arr_triple(int * a, size_t n) {
    int * p;
    for (p = a; p < a + n; p++)
        *p *= 3;
}
```

Using function pointers:

```
void arr_apply(const int * a, size_t n, void(*f)(int)) {
    const int * p;
    for (p = a; p < a + n; p++)
        f(*p);
}
```

Printing the array:

```
void print(int n) {
    printf("%d\n", n); }
int a[] = {3, 2, 7, 6, 8};
    arr_apply(a, 5, print);
void arr_process(int *a, size_t n, void(*f)(int *)) {
    int *p;
    for (p = a; p < a + n; p++)
        f(p);
}
```

Tripling the integersinthe array

```
void triple(int * p) {
    *p *= 3;
}
int a[] = {3, 2, 7, 6, 8};
arr_process(a, 5, triple);
```

*p++, *++p, ++*p, ...

* ++p  (*++)p  or  * (++p)  * ++p == *(++p)

(*++)p doesn't make sense

++*p  (++*)p  or  ++(*p)  ++*p  == ++(*p)

*******are we changing the pointer or are we changing the object?

*p++  (*p)++ or *(p++) ?  from the precedence table (with difficulty)

we see that *p++ == *(p++)

It turns out the most common version --> it allows us to process the current object and then make p point to the following object

NOTE: you can't increment or decrement a void pointer

**Example 1**

```
int a[] = {1, 3 ,5};
int *p = &a[1];
printf("%d", *++p);
printf("%d", *p);       /* prints out 55 */
```

**Example 2**

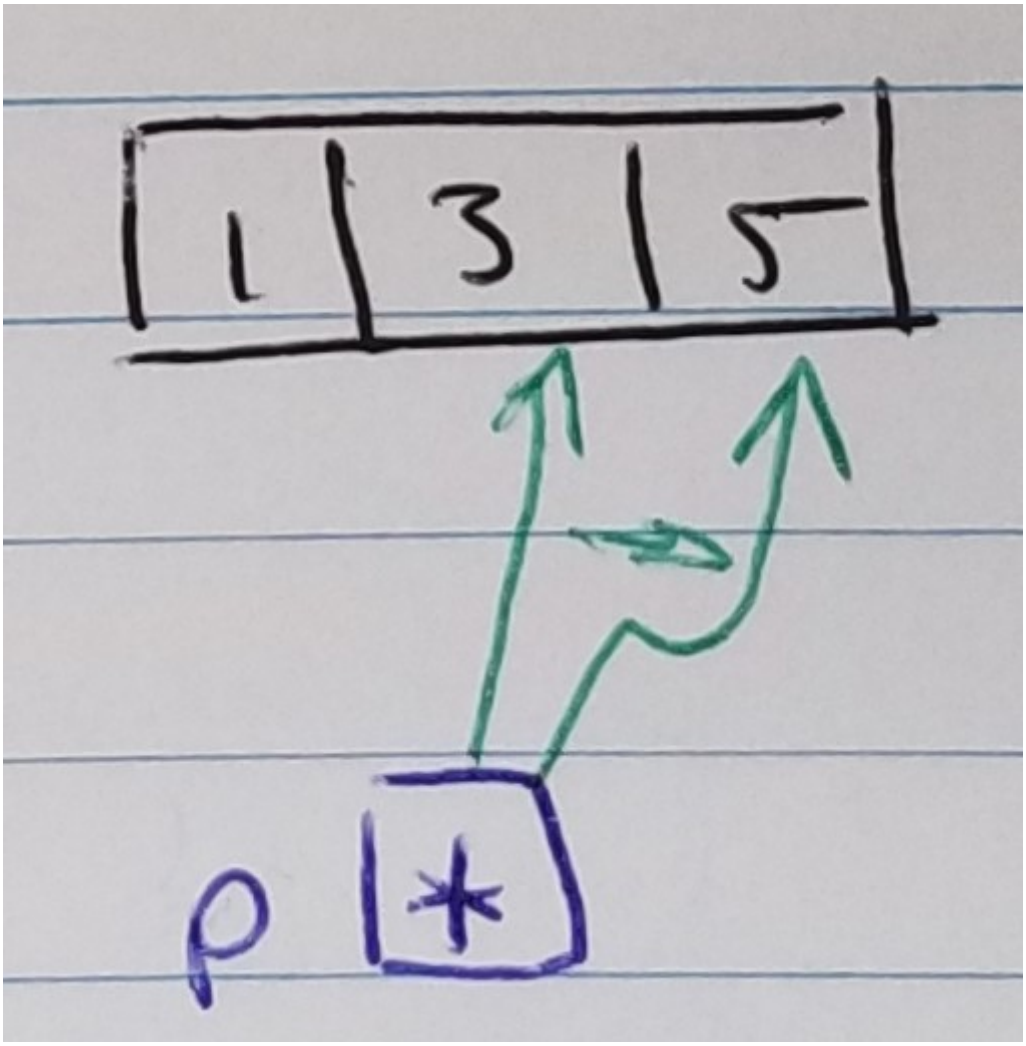
```
int a[] = {1, 3 ,5};
int *p = &a[1];
printf("%d", ++*p);
printf("%d", *p);       /* prints out 44 */
```

**Example 3**

```
int a[] = {1, 3 ,5};
int *p = &a[1];
printf("%d", *p++);
printf("%d", *p);   /* prints out 35 */
```

**Example 4**

```
int a[] = {1, 3 ,5};
int *p = &a[1];
printf("%d", (*p)++);
printf("%d", *p);   /* prints out 34 */
```

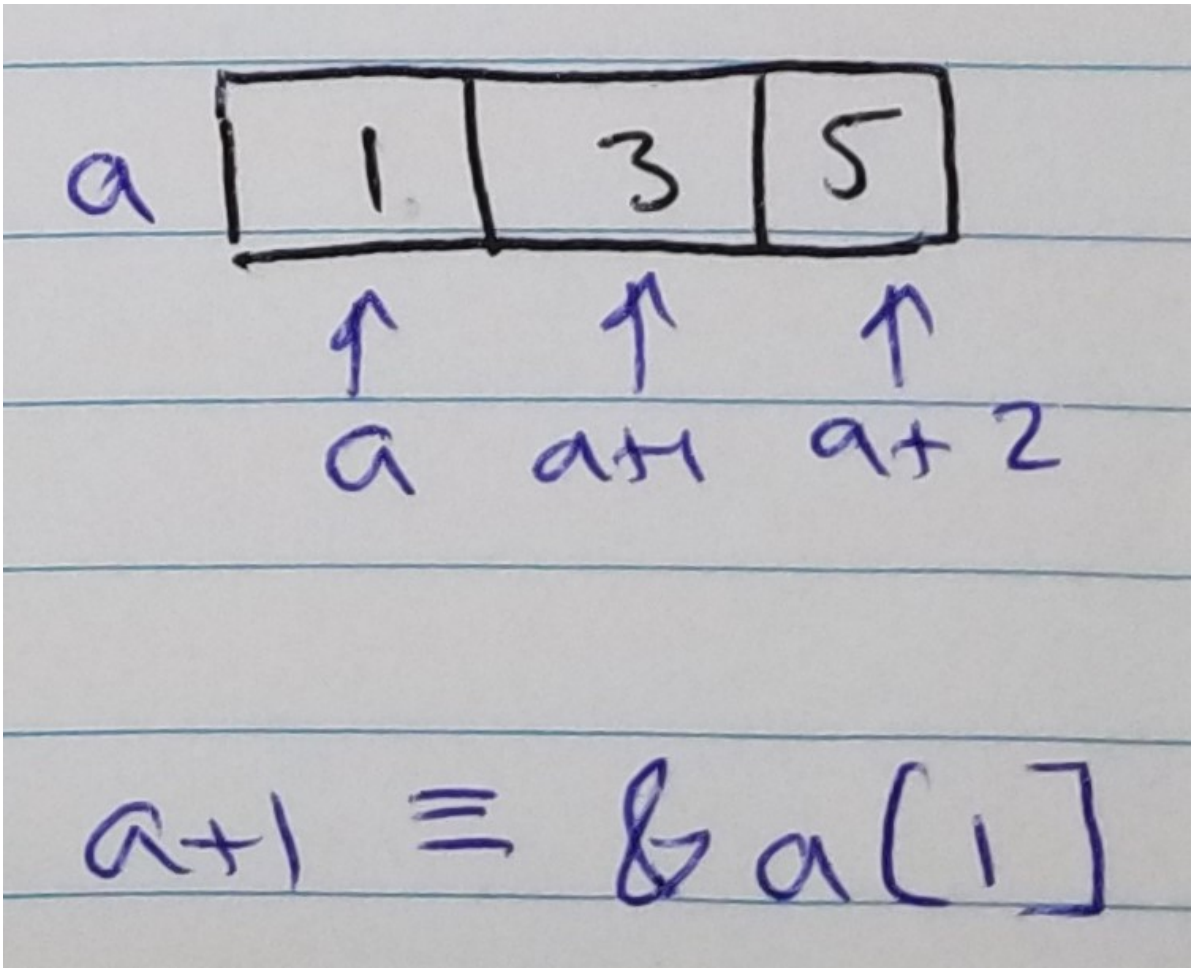



## Equivalence of pointer and array notation

```
* (x + n) == x[n]
x + n == & x[n]
```

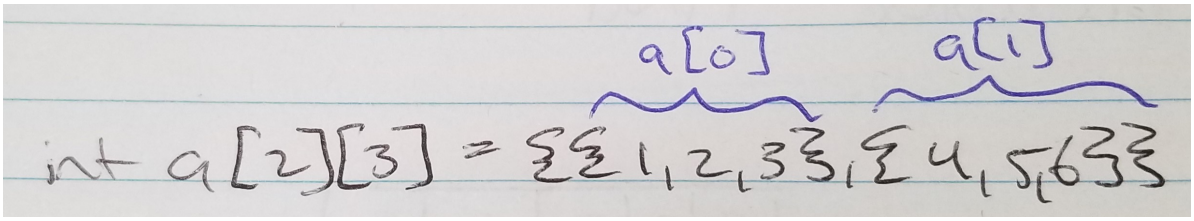
for x either an array or a pointer

```
int a[] = {1,3,5};
int *p = &a[1 ];        /* or */    int *p = a+1
```




**Example**

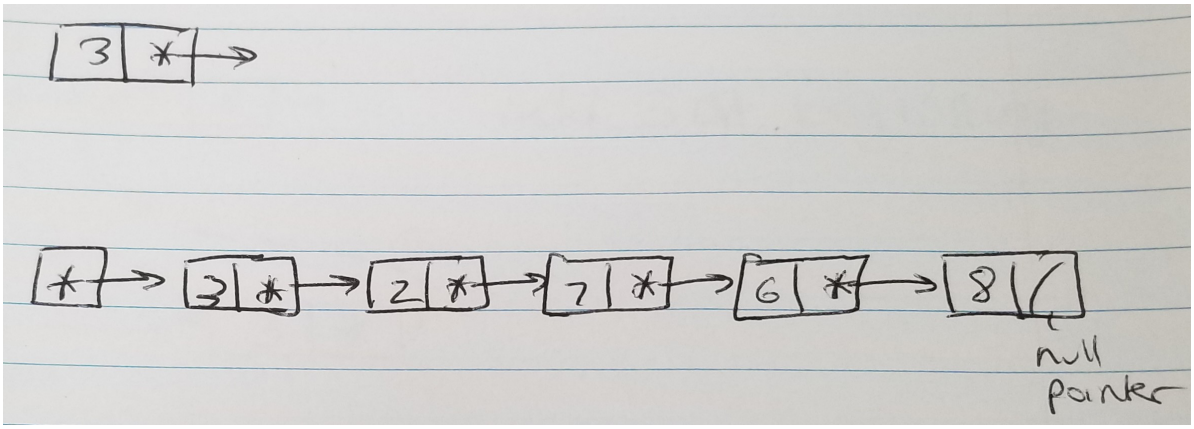
```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};

a[0][1] == (*a)[1] == *((*a) + 1) == *(a[0] + 1)

a[1][2] == (*(a + 1))[2] == *(*(a + 1) + 2) == *(a[1] + 2))
```




# Singly Linked Lists

A linear sequence of nodes with each node (except for the last node) containing a pointer to the following node

A list is represented by a pointer to the first node (head pointer)




**Example: list of integers**

```
typedef struct node node;
struct node {
    int data;
    node *next;
}
```

## Operations on Lists

1) adding data to a list

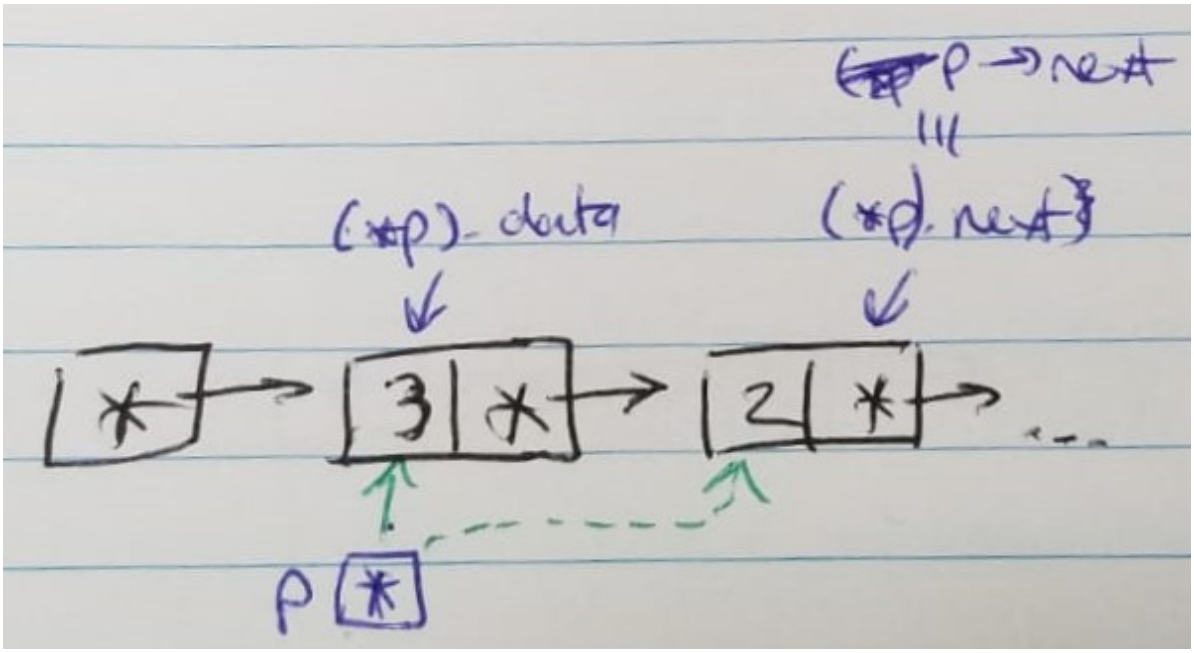
2) removing data from a list

3) destroying a list

4) traversing a list

**Example 1 - Traversing a list**

REMINDER: given 2 pointers p and q, p= q makes p point to the same thing as q




how do i make p point to the following node

```
node *p;
for (p = head; p != 0; p = p->next)
```

POSSIBLE EXAM QUESTION!!!!!

```
"we have a string"

char *s

camel_case(char *s) {
    int upper = 1;
    while (*s) {
        if(upper)
            *s = to_upper(*s);
            upper = 0;
        }
        if (*s == ' ') {
            upper = 1;
            s++;
        }
}
```

# Standard idiom to process a list

(Single pointer version)

```
/* head is the head pointer */
node *p;
for (p = head; p != 0; p = p->next){
    /* process p -> data */
}
```

**Example 1 - Summing the integers in a list of integers**

```
int list_sum(node * head) {
    node * p;
    int sum = 0;
    for (p = head; p != 0; p = p->next)
        sum += p->data;
    return sum;
}
```

**Example 2 - Looking for an integer in a list of integers**

```
node * list_find(node * head, int data) {
    node *p;
    for (p = head; p != 0; p = p -> next)
        if(p->data == data)
            return p;
    return 0;
}
```

# November 29, 2018

## Destroying a list

Each node is dynamically allocated and we need to free each node

We can't use the standard idiom to traverse a list to do this

once we perform free(p), we can't access p->next

FIX: remember p-> next before we call free(p)

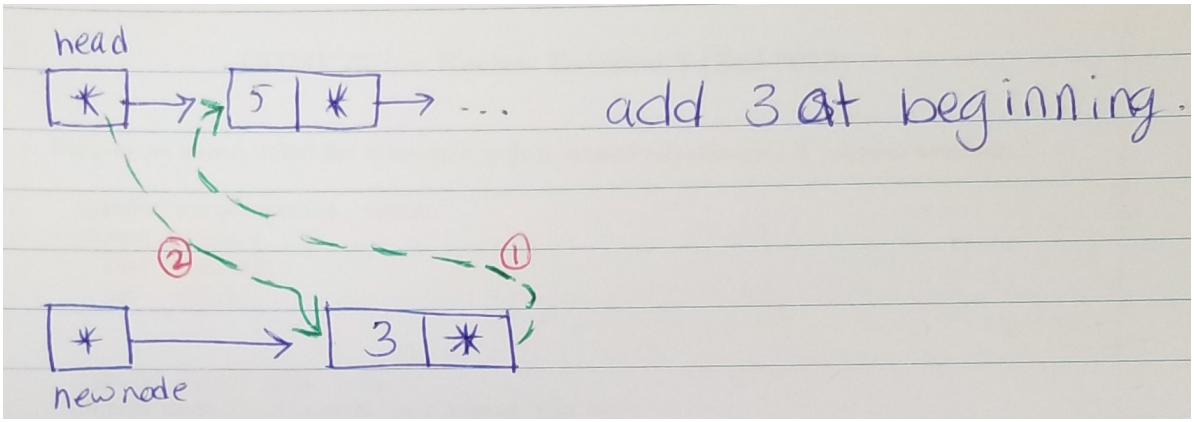
# Standard idiom to free a list

```
head -pointer
node *p, *q;
for (p = head; p != 0; p = q) {
    q = p -> next;
    free(p);
}
```

# Adding data to a list

We'll look at 2 versions:

**Version1 - adding at the beginning of the list**




add 3 at beginning

steps:

1) create a new node and store 3 in it

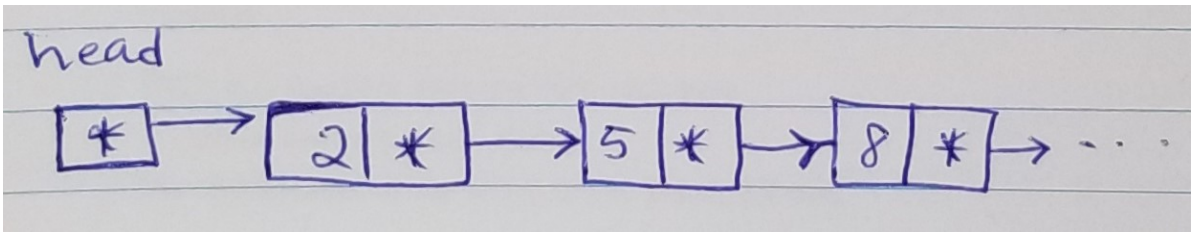
2) insert new node at beginning

```
node * newNode;
newNode = malloc (sizeof(node));
if(newNode = 0) {   /* malloc failed */
    /* error handling */
}
/* assume malloc succeeds here */
newNode->data = 3;
newNode->next = head;    /* 1 */
head = newNode;          /* 2 */
```

**Exercise: encapsulate the adding of data into a function**

```
list_insert_head(node **phead, int data);
```

**Version 2 - adding data in some sorted order**




add 7 - we need to find the correct location to insert the new node

using the previous standard idiom to traverse the list in doesn't quite work

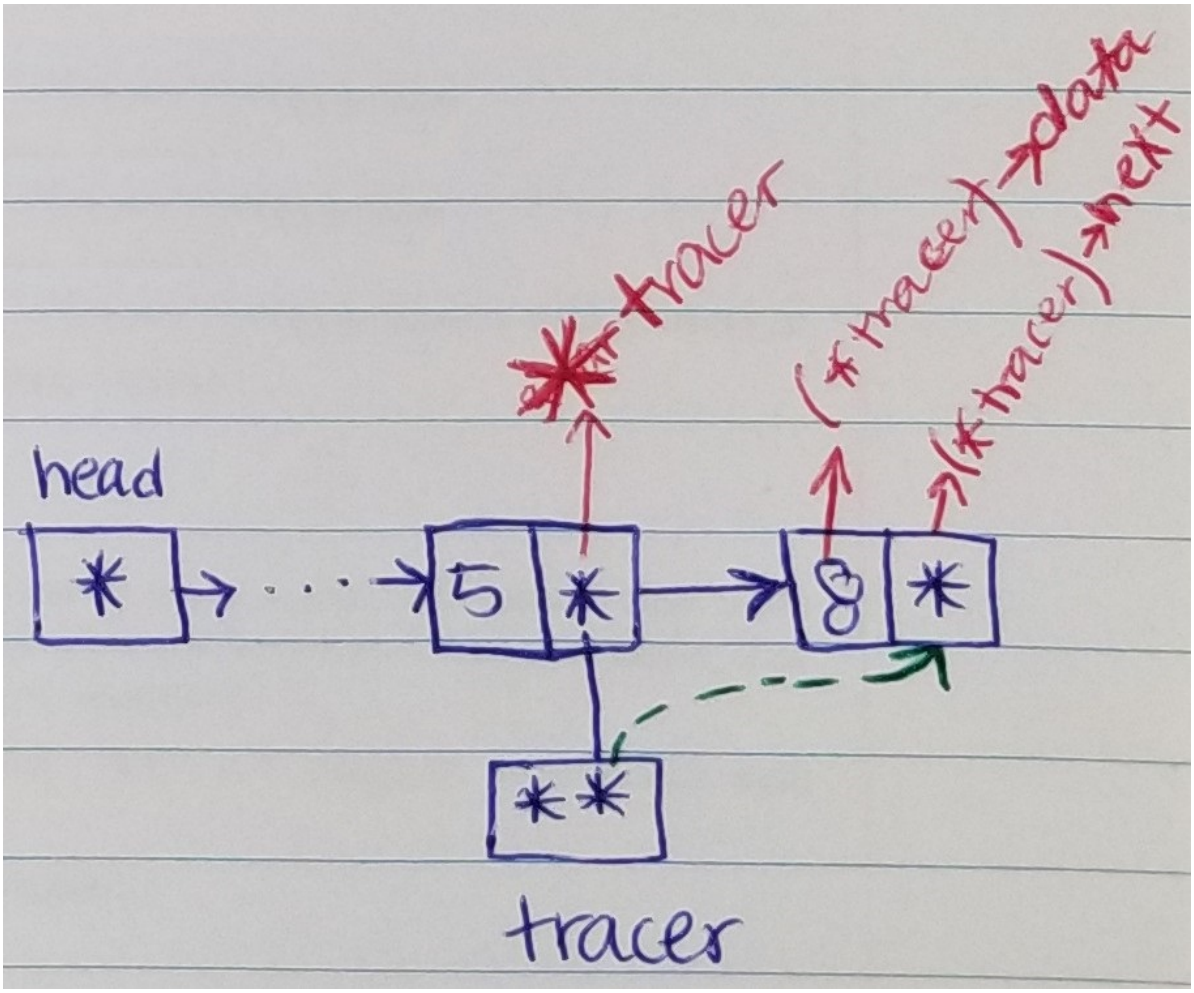
(it would start at the node containing 8)

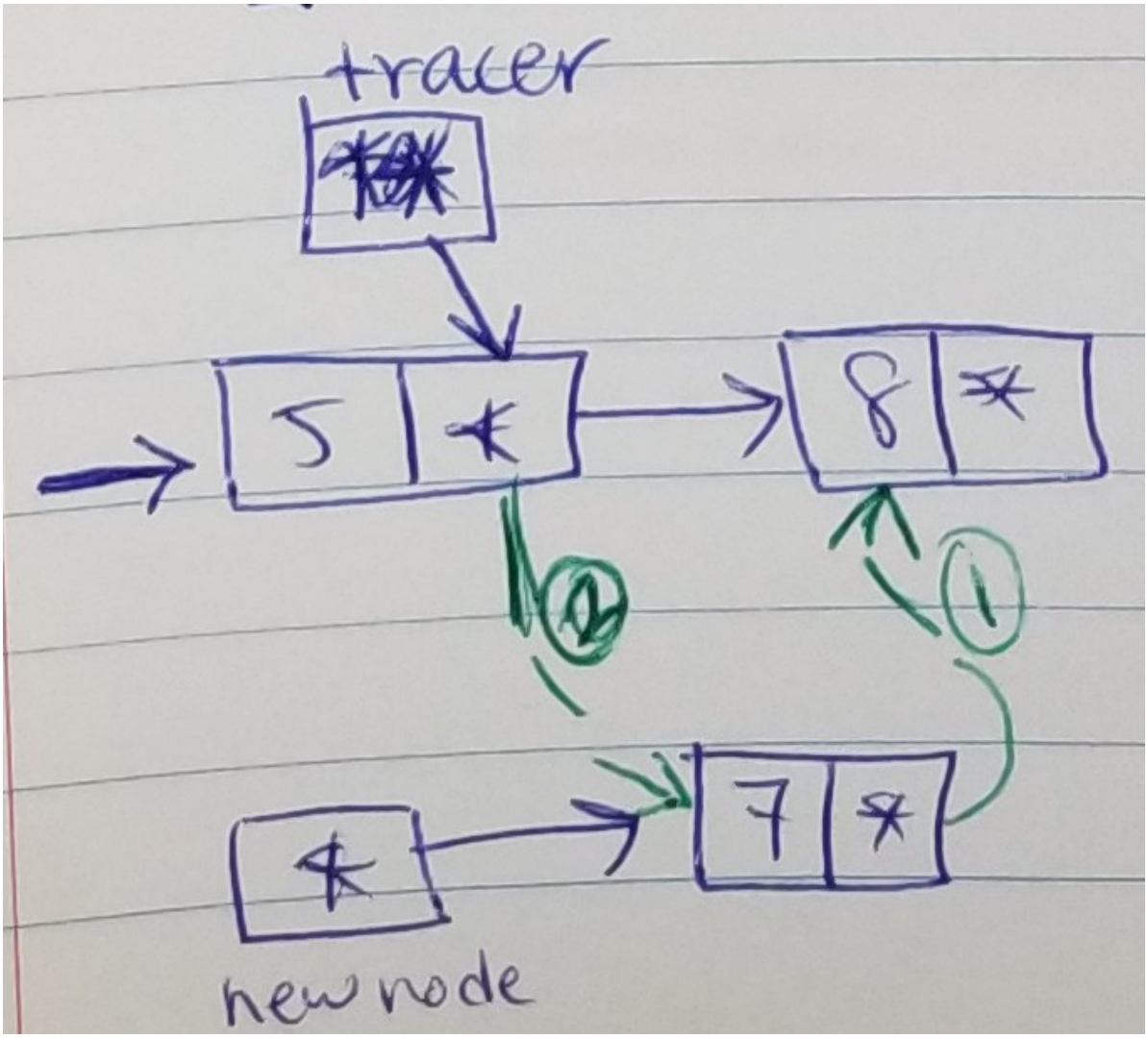
2 solutions:

1) use a pair of pointers to traverse the list

2) use a double pointer

We will use 2





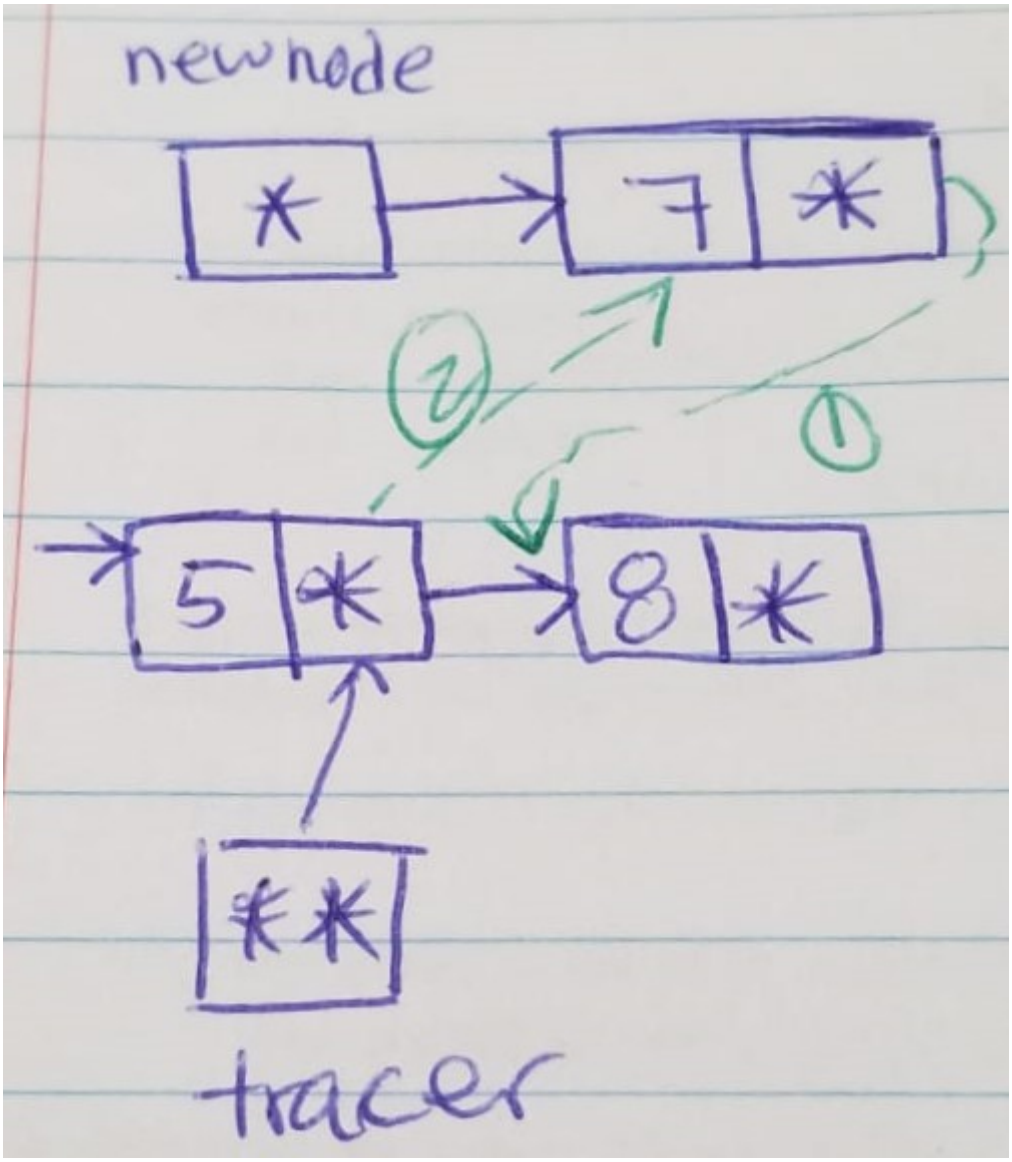

```
    /* tracer = &head */
for(tracer = &head; *tracer != 0; tracer = &tracer->next)
```

## Standard idiom to traverse a list

head - head pointer

```
node **tracer;
for(tracer = &head; *tracer != 0; tracer = &(*tracer)->next)
    /* process (*tracer)->data */
```

NOTE: this version may be needed when we add and remove data, if we are adding at the beginning we don't need this, we do if in some particular order.  Otherwise use a single pointer version.




```
int list_insert_sorted(node **phead, int data) {
    node *newNode;
    node **tracer;
    for(tracer = phead; *tracer != 0; tracer = &(*tracer)->next)
        if((*tracer)->data >= data)
            break;
    newNode = malloc(sizeof(node));
    if (newNode == 0)
        return 0;
    newNode->data = data;
    newNode->next =
    *tracer = newNode;
    return 1;
}
```

# December 5, 2018

## Removing data from a list

img1

Need to use a double pointer to traverse the list

make the double pointer skip the target node and point to the next node beyond it

```
*tracer = (*tracer)->next   /* brackets otherwise dereferencing ->next as well
*/
```

```
list_delete(node **phead, int data) {
    node **tracer;
    for(tracer = phead; *tracer != 0; tracer = &(*tracer)->next)
        if((*tracer)->data == data)
            break;
    /* assert(*tracer == 0 || (*tracer)->data == data); */
    /* never dereference a null pointer unless you know for sure it's not null
*/
    if(*tracer != 0) {
        node *tmp = *tracer;
        *tracer = tmp->next;    /* skip node */
        free(tmp);
        return 1;
    }
    return 0;
}
```

*IMPORTANT: never dereference a null pointer unless you know for sure it's not null*

Printing a singly linked list backwards using recursion

```
void reverse_print(node *head) {
    if (head != 0) {
        reverse_print(head->next);
        printf("%d\n", head -> data);
    }
}
```

Print a linked list forward

```
void print(node *head) {
    if(head != 0) {
        printf("%d\n", head -> data);
        printf(head -> nest);
    }
}
```

# Binary Search Tree

img2

Each node has 2 subtrees (which may be empty)

Left side is less than or equal to, right side is strictly greater than

  some structures don't allow equal to

Each node has a key, and at each node, the keys in its left subtree are smaller (or equal) than the key in that node; the keys in the right subtree are bigger than the key in that node

## Three ways to traverse a tree

pre order:  current node, left subtree, right subtree

in order:  left subtree, current node, right subtree

post order:  left subtree, right subtree, current node

```
typedef struct node node;
struct node {       /* node of a binary tree */
    int data;
    node *left, *right;
};
node *root = 0;     /* empty tree; the root pointer represents the tree*/
```

```
void preorder(node *tree) {
    if (tree !=0)
        printf("%d\n", tree -> data)
        preorder(tree -> left);
        preorder (tree -> right);
}
```

in order and post order are similar

# Destroying a binary tree

Involves traversing a tree and freeing each node

must use post order traversal because once we free the node we can't access the other side of the node

```
void destroy(*node *tree) {
    if tree != 0) {
        destroy(tree -> left);
        destroy(tree -> right);
        free(tree);
    }
}
```

img3

preorder: 100, 80, 70, 75, 90, 120, 110

in order: 70, 75, 80, 90, 100, 110, 120

post order: 75, 70, 90, 80,110, 120, 100

# Looking for an item in a binary search tree

```
/* should probably use 'leaf' instead of tree */
node *find(node * tree, int data) {
    if(tree == 0)
        return 0;
    if(tree->data == data)
        return tree;
    if(data < tree -> data)
        return find(tree -> left, data);
    if(data > tree -> data)
        return find(tree -> right, data);
}
```

# Inserting data into a binary search tree

```
int insert(node **ptree, int data) {
    if(*ptree == 0) {
        *ptree = malloc(sizeof(node));
        if (*ptree == 0)
            return 0;
        (*ptree) -> left = (*ptree) -> right = 0;
        (*ptree) -> data = data;
        return 1;
    }
    if (data < (*ptree) -> data)
        return insert(&(*ptree) -> left, data);
    if (data > (*ptree) -> data)
        return insert(&(*ptree) -> right, data);
    return 0;       /* don't allow duplicates in this case */
}
```

**Example: finding the maximum value in the binary search tree of integers**

```
int max(node *tree) {           /* precondition: tree != 0 */
    while(tree -> right != 0)
        tree = tree -> right;
    return tree -> data;
}

int max_v2(node *tree) {
    if (tree -> right == 0)
        return tree -> data     /* precondition: tree != 0 */
    return max_v2(tree -> right);
}
```

**Example**

img 4

insert 85, 60, 90, 120, 35, 50, 110 into an empty binary search tree

preorder: 85,60, 35, 50, 75, 90, 120, 110

post order: 50, 35, 75, 60, 110, 120, 90, 85

# FINAL

17 short questions, 17 marks for writing code

array vs pointer 2 to 3

right left rule 2 to 3

combined 5 to 6

comparison function qsort 3

linked list 4

binary tree 4

need to use malloc where appropriate

no questions on preprocessor

don't need to focus on fscanf sscanf and file open