September 5, 2018

General about C

Created by Dennis Ritchie in the early 1970s

Created to implement UNIX

Procedural as opposed to object oriented --> function based

We'll be using ANSI C and the gcc compiler

Example: Hello world:

hello.c --> file name ends with '.c'

only one kind of comment --> /* text */

```
#include<stdio.h>

int main(void) {
    printf("hello world!\n");
    return 0;
}
```

int is the return type.

main is the name of the function

Contents of brackets are parameter list

NOTE: void means doesn't take any arguments

Main defines where to start executing, a prime mover function

Value returned by main is passed to the shell that starts the program

there can only be one main function

ANSI C specifies two valid versions of main

1. int main(void)
2. int main(*something other than main*)

CONVENTION: main should return 0 if there are no other errors

otherwise it returns a positive integer

Compiling and running the program

download and install www.cygwin.com

be sure to install gcc

$ is the prompt like C:

```
$ gcc -ansi -W -Wall -Werror -pedantic hello.c
```

-ansi means use ncc

-W means turn on warnings

-Wall means turn on all warnings

-Werror means if there are warnings regard them as error

-pedantic means strict ncc

if successful this generates a file named a.exe (in cygwin)

$ ./a (Cygwin)

$ ./a.out (Linux/Mac)

The return value of main is passed back to the shell and the shell can examine that value

echo %errorlevel% (Windows)

echo $? (bash shell)

when you log in system creates a bash shell

very first program must be specially created (with init)

Shell gets whatever the return value of main

Example: square.c --> Calling our own function from main:

```
#include <stdio.h>           /* read in the file sdio.h */
                             /* Like java import
                             /* stdio means standard i/o */

int square(int n);
                             /* declaration of square type */
int main(void) {
    int n = 13;
    prinf("%d\n", square(n));       /* %d is to print an integer in base 10 */
    return 0;                       /* to indicate it ran properly */
}

int square (int n) {
    return n*n;
}
```

must declare square before trying to use it or compiler will complain (function prototype)

compiling this program will give you an error b/c square is called before the creation (reads from top to bottom)

Declaration vs Definition:

Definition gives the actual code

definition is also a declaration

declaration is the existence of function

declaration only tells compiler existence of function, what it takes and what it returns

definition gives the actual implementation of the function

NOTE: a definition can serve as a declaration

a declaration allows the compiler to check whether we are using the function correctly

NOTE: the function prototype for printf is in stdio.h

Basic Printing:

NOTE: printf needs to take in a string --> cannot do printf(n);

```
printf("%d", n);
```

%d is for integer base 10

%0 is for unsigned integer in base 8

%x, %X for unsigned integer in base 16

%f is for float or double

%s is for string

%c is for a character

%u is for unsigned complement

Arrays:

Examples:

```
int a[5];
```

array of 5 integers

NOTE: There is no such thing as null in C

Can create global variables in C

most of the variables we use will be local though

Contains whatever happened to have been there in memory if undefined

bits not randomized or overwritten

NOTE: Forgetting to initialize variables is a danger in C

Valid elements: : a[0] a[1] a[2] a[3] a[4]

```
int b[5] = {3, 2, 7, 6, 8};
```

initialized array instantiation

a[0] is 3, etc

```
int c[5] = {3, 2, 7, 6, 8, 1};
```

not allowed; too many initializers

```
int d[5] = {3, 2, 7};
```

not enough initializers, the rest default to zero

```
int e[] = {3, 2, 7, 6, 8};
```

compiler counts the number of initializers

assuming 32 bit ints, the size of the array e is 20 bytes

Standard idiom to process an array:

```
t a[n];

size_t i;
for (i = 0; i < n, i++)
```

t stands for the name of some type, any type

n stands for some positive integer

works for any kind of array

in ANSI C, variable must be declared at the beginning of a block*

Example:

```
int main (void) {
    int x = 1;
    printf("%d\n", x);
    int y = 2;              /* error: not the start of a block */
    if (x > 0) {
        int z = 3;          /*OK, because new block */
    }
    int z = 4;              /* error: not the start of a new block */
}
```

size_t:

It's the name of the type. TYPE to store sizes

It's some unsigned integer type

Only non negative

We are returning -1 if we can't find the integer in the array

NOTE: -1 has the same bit pattern as the biggest possible value under two's complement

size_t is an unsigned integer of at least 16 bits, therefore it's a data type that is guaranteed to hold any array index

It is the type to store size

It is common convention to name something underscore t

We want to encapsulate the summary of the array in a function

Examples:

1) Summing an array of integers:

Passing an array into a function is complicated --> later section

```
int main(void) {
    int a[] = {3, 2, 7, 6, 8}
    size_t i;
    int total = 0;              /* must be initialized */
    for (i=0; i<5; i++)         /* don't need braces if only one line */
        total += a[i];
    printf("%d\n", total);
    return 0;
}
```

NOTE: cannot do arr.length because that is object oriented programming

September 6, 2018

pass-by-value vs pass-by-reference

When we pass an object into a function, does the function get a copy (*pass-by-value*) or the original object (*pass-by-reference*)?

In C, essentially everything is passed by value.

```
void triple(int n) {
    n *= 3;
    int n = 2;
    triple(n);
}
```

n is still 2 after this, function gets a copy of n

However, it is impossible to pass a whole array by itself into a function.

When we pass the name of an array into a function, *the function gets the starting address of the array, it doesn't get the whole array*.

Any function that processes an array will need at least 2 parameters:

the starting address which is the array name

the number of elements in the array

2) function to sum an entire array:

```
int arr_sum(const int a[], size_t n) {
    int total = 0;
    size_t = i;
    for ( i=0; i<n; i++)
        total += a[i];
    return total
}
```

NOTE: const here tells compiler that this function does not change the elements in the array

--> It is the c equivalent of 'final'

NOTE: size_t n is the number of elements

3) find maximum value of an integer array:

```
int arrr_max(const int a[], size_t n) {     /* precondition: n>0 */
    int max = a[0];
    size_t i;
    for (i=0; i<n; i++)
            if (a[i] > max)
                max = a[i];
    return max;
}
```

September 12, 2018

REMINDER: In C, all variables must be declared at the beginning of a block

```
void f(void) {
    int i = 1;
    i++;
    int j = 2;      /* NOT ALLOWED!!! */
}
```

Design by Contract (DBC):

A function can be viewed as a contract between 2 parties:

the implementers of the function and the users of the function

printf will print certain things if another party provides input to print f

each party must satisfy certain conditions, both have obligations

precondition: the condition the caller/user of the function needs to satisfy in order to have the function "work correctly" (ie it does what it promises to do)

postcondition: what the function promises to do if the precondition is met

making preconditions and postconditions explicit can reduce bugs.

What's the difference between a bug and an error?

they are both exceptional circumstances

an *error* is something that should never happen

a *bug* is the result of a failure to meet a precondition

The *assert macro* is used to check preconditions for debugging purposes

#include <assert.h> required at beginning of program to perform these checks

```
int arr_max(const int a[], size_t n) {          /* precondition: n>0 */
    int max;
    size_t i;
    assert (n>0);
    max = a[0];
    for (i=0; i<n; i++)
        max = a[i];
    return max;
}
```

NOTE: If the assertion fails, the assert macro terminates the program, printing the filename and the line number of the assert macro

Assert is for debugging, not error handling

NOTE: When we finish debugging, we can turn off assert by defining the NODEBUG macro

The easiest way to specify this when compiling the program:

gcc -ansi -W -Wall -Werror -pedantic -DNDEBUG lab.c

will turn off the debug and it won't run anymore --> don't need to actually delete the assert line from the program

4) looking for an integer in an integer array:

```
size_t arr_find(const int a[], size_t n, int x){
    size_t i;
    for (i=0; i<n; i++)
        if (a[i] ==x)
            return 1;
    return -1;          /* returns the biggest possible value for size_t */
}
```

5) tripling the integers in an integer array:

```
void arr_triple(int a[], size_t n) {
    size_t i;
    for (i=0; i<n; i++)
        a[i] *= 3;
}
```

Testing arr_max:

1) print return value of array_max (*not a good way*):

```
int a[] = {3, 2, 7, 6, 8};
printf("%d\n", arr_max(a,5);
```

print the return value of arr_max

2) print whether the function gives the correct answer:

```
int a[] = {3, 2, 7, 6, 8};
printf("%d\n", arr_max(a,5) == 8);
```

print whether the function prints the correct answer, easier to see if it passed or failed

printf(%d\n") is sufficient to because we are dealing with an integer 0, or 1

3) print the test as well as whether it succeeds or fails (*preferred*)

use the following macro

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

REMINDER: macros are the # things we put at the top of the program and at the command line

REMINDER: ternary functions --> a ? b : c --> if a, then b, if not then c

```
int a[] = {3, 2, 7, 6, 8};
CHECK(arr_max(a, 5)) =);
```

test passes --> arr_max(5) == 8

NOTE: there is no Boolean type in ANSI C. 0 (or its equivalents) is false (even -1); everything else in time...

EXAMPLE: It may be useful to group tests into functions

```
void arr_max_test(void) {
    int a[] = {3, 2, 7, 6, 8};
    /* some lines of relvant code */
}

void arr_find_test (void) {
    /* some lines of relvant code */
}

void main(void) {
    arr_max_test();
    arr_find_test();
}
```

September 13, 2018

Strings:

There is no separate string type in C

A string is simply an array of characters terminated by the null character. The null character can be denoted by '\0' (or simply 0 because the ANSI C code for NULL is 0)

Examples of Strings:

1) "hello" --> this is a string constant

Any attempt to change it's characters leads to undefined behaviour

We say that the length of this string is 5, 5 characters

But we need an array of at least 6 characters to store it as a string, because strings are terminated by the null character

NOTE: We don't need to explicitly write the null character (it already has this when presented as a string)

NOTE: length does not include null character

2) an array of 6 characters:**

```
char s[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

3) the same as writing out #2

```
char t[] = "hello";
```

Standard idiom to process a string:

NOTE: the null '\0' character isn't processed

```
/* assume 's' is a string */

size_t i;
for (i=0; s[i] != '\0'; i++)
    /* process s[i] */
```

```
char s[] = "hello world";
printf("%s",s);                 /* hello world */
s[4] = '\0';
printf("%s",s);                 /* hell */
```

NOTE: when we change index 4 to NULL, we truncate the string, but the original array holding all of the characters still exists, only with the index 4 having been changed from 'o' to '\0'

Examples of processing strings:

1) length of a string

```
size_t str_length(const char s[]) {
    size_t i;
    for(i=0; s[i] != '\0'; i++);
    return i;
}
```

i will have incremented to the length of s[]

2) changing a string to uppercase:

NOTE: must include #include <ctype.h> to use tolower/toupper/isalpha/isdigit/isalnum --> it is a library which includes these functions

```
#include <ctype.h>

void str_uppercase(char s[]) {
    size_t i:
    for (i=0; s[i] != '\0'; i++)
        s[i] = toupper(s[i]);
}
```

str.uppercase("hello") is BAD! it can lead to undefined behaviour

str.uppercase(h) is GOOD!

NOTE: toupper() is passed by value, not by reference. This means that toupper() doesn't change the string, it only returns the uppercase ASCII value.

REMINDER: void means doesn't take any arguments

3) looking for characters in a string:

NOTE: 'a' has type int in C, because everything is reduced to an integer in ASCII code

```
str_find(const char[s], int x) {}
    size_t i;
    for (i=0; s[i] != '\0'; i++)
            if (s[i] == x)
                return i;       /* returns index of character found */
    return -1;              /* returns negative response if not found */
}
```

4) making a copy of a string

NOTE: dest[] isn't const char because we are going to be overwriting an empty array with characters from the source array, we can't have it be constant.

```
str_copy(char dest[], const char.src[]) {
    size_t i;
    for (i=0; src[i] != '\0'; i++)
        dest[i] = src[i];
    dest[i] = src[i];  /* or: */ dest[i] = '\0'; */
}                      /* both ways of copying ending null character */
```

NOTE: the programmer is responsible for ensuring that the destination array is large enough to store the src string to avoid buffer overflow!! --> C doesn't have an out-of-bounds exception

September 19, 2018

Bugs & Errors

C is a dangerous language --> even if program runs a million times correctly, there is no guarantee that it'll run correctly the next time!

2 common sources of bugs:

1) Uninitialized local variables:

Any uninitiated local variables will be assigned random (whatever was already in that memory location) so no two uninitiated variables will be equal

Don't forget to initialize a variable before you use it --> such as forgetting to initialize the sum to 0 when summing an array

2) Overflowing a buffer --> widening outside an array

There are no warnings if an overflow occurs, it will simply overwrite data beside it in memory

example --> storing too many values in an array

Testing can show that there are bugs, but it cannot show that there *aren't* any bugs

Some string functions in the standard C library:

1) find the length of string: strlen

```
strlen("hello");                    /* (!dangerous!) */

size_t strlen(const char s[]);
```

2) function to copy a string: strcpy

```
char s[100] ;
strcpy(s, "hello");
```

's' is the destination, "hello" is the source

Note: programmer is responsible for ensuring the destination array is large enough to store the source string!

need to hard code the size

Note: We cannot create an array of variable size in ANSI C. Example:

```
int n = 100;
int a [n];  /* NOT ALLOWED */
```

NOTE: in general, you will need to code a large buffer because we can't use a variable to code the size of an array, or a safer version is to use strncpy(destination, source, # to copy), but strncpy isn't easy to use

Safer/Correct version of string copy: strncpy

```
char s[100];
strncpy(s, source, 99);     /* could specify 100 and would work */
dest[99] = '\0';            /* this line of code is crucial */
```

s = destination, source = source string, 99 = number of characters to copy

In case destination is not large enough, need to manually add a null character. One less character because last character is set to the terminator

NOTE: we do this in case dest[] isn't large enough, we need to manually add NULL character

```
char s[6];
strncpy (s, "goodbye", 6);

/* BAD!  left with 'goodby' because there is no terminating character, we need to
manually add a terminating character or the program will keep running
indefinitely */

s[5] = '\0';        /* s = "goodb'\0'"
```

3) sting comparison: strcmp

strcmp(s1, s2) --> returns a negative integer if s1 is less than s2, 0 ir s1 is equal to s2, and a positive integer if s1 is greater than s2 (in lexicographic order)

```
strcmp("hell", "hello")  /* is negative */
strcmp("hello", "hell")  /* is positive */
```

example: test str_uppercase -->

```
char s[]="hello";
str_uppercase(s);
CHECK(strcmp(s, "HELLO") == 0);
```

Command-line Arguments

```
$ gcc -ansip-W -Wall -Werror -pedantic lab2.c
```

Everything after $ are called command line arguments, and passed to the program via main)

there is a 2nd version of main:

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

In the above example, argc is 7 and argv is the array of strings of the command line arguments

argv[0] is "gcc"

argv[1] is "-ansi"

argv[2] is "-W"

argv[3] is -Wall"

argv[4] is "-Werror"

argv[5] is "-pedantic"

argv[6] is "lab2.c"

This is the main needed for when you need to know the command line arguments

```
#include <ctype.h>

void str_uppercase(const char s[]) {
    size_t i:
    for (i = 0; s[i] != '\0'; i++)
        s{i} = toupper(s{i});
}

int main(void) {
    str_uppercase("hello");
    return 0;
}
```

hello is a string constant, this will not work

Standard way to process command line arguments:

```
int main (int argc, char * argv[]) {
    int i;
    for (i = 1; i < argc; i++)
        /* prcocess argv[i] */
}
```

Example --> the echo program:

```
$ echo hello world
hello world
```

does echo need to print back its own name? No, because it's processed

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;

    for (i = 0; i < argc; i++) {
        printf("%s ", argv[i]);         /* after "world" there is always a space
*/
    }
    print("\n");
    return 0;                           /* why zero?????????? */
}
```

Fix 1:

```
for (i = 1; i < argc; i++)
    if (i == argc -1)
        printf("%s\n", argc[i]);
    else
        printf("%s ", argv[i]);
```

Fix 2:

```
for (i = 1; i < argc; i++)
    printf(i == argc -1 ? "%s\n" : "%s ", argv [i]);
```

Fix 3

```
printf("%s%c", argv[i], i == argc -1 ? '\n' : ' ');
```

Input/Output

I/O is performed via streams

Output

When the computer starts, 3 streams are already available

1. Standard input (stdin)

   Associated with the keyboard input by default

2. Standard output (stdout)

   Associated with the console output by default

3. Standard error (stderr)

   Associated with the keyboard input by default

We can change these associations using I/O redirection from the shell

$ (prompt) .\a (program) < input > (redirects standard out) output 2> (redirects stderr) error

input, output, and error are filenames

```
$ .\a < input > output 2 > error
```

CONVENTION: regular messages are printed to standard output, error messages are printed to standard error

September 20, 2018

Output: 3 related functions

printf / fprintf / sprintf

printf prints to standard output (stdout)

fprintf prints to a stream that we can specify

sprintf to a string (an array of characters)

```
int n = 1, m = 1;
    printf("The sum of %d and %d is %d\n", n, m, n+m);
```

"the sum of d and d is d" is a format specification / conversion specification

In ANSI C, the conversion specification is started by a % and can consist of up to five components

```
slong double f = 1.3456;
printf("%08.3lf", f)        /* 0012.245 */  /* 'L' is case sensitive */
```

0 --> flag (pad with zeroes)

8 --> minimum field width (minimum width of 8)

.3 --> precision (decimal places

l -->modifier (modifies the f to print a long double)

f -->specifier (float or a double)

output: 12.346

When you would need a modifier:

```
long n = 123;
printf("%d", n);
printf("%5ld", n);      /* __123 */
```

modifier d to print a long int

fprintf: basically the same a sprintf except that it takes an extra argument (the 1st argument) that specifies the stream to print to

```
long n = 123;
fprintf(stderr, "%5ld", n);

basically...
    printf(...) == fprintf(stdout, ...)
```

CONVENTION: Error messages should be printed to stderr

sprintf: basically the same argument except that it takes an extra argument (the 1st argument) that specifies the array of characters to point to

```
char a[100];
long n = 123;
sprintf(a, "%ld", n);  /* now contains the string "123" */
```

sprintf is dangerous!! --> it can overflow the destination buffer!!

some other output functions:

1) putchar(c); for printng a character(to stdout)

2) puts(s); character (type int) --> for putting a string to (stdout) followed by new line character

puts(s, "hello world"); as opposed to printf("hello world\n")

Input

More complicated than output as input can frequently fail

IMPORTANT: always check the return value of an input function immediately after it returns

Input can be performed one of 4 ways:

1) character by character

```
int c;
while ((c=getcharC())!=EOF)
        putchar(c);
```

getchar() returns EOF when it fails (end-of-file or read error)

what is EOF? It is the name of the value returned by getchar() when it fails

c is declared as an int

c is declared as an int because getchar() returns all possible character values as well as EOF to indicate failure, hence its return type must be bigger than a char.

EOF is typically defined to be -1)

2) line by line

3) item by item

4) block by block

September 27, 2018

```
int c;
while ((c=getchar()) != EOF) putchar(c);
```

The brackets around c = getchar() are necessary because "!=" has higher precedence than "=" NOT EQUAL operator.

What is End-Of-File?

It is a condition. For a file on disk, end-of-file becomes true if we try to read *past* the last byte in the file.

For keyboard input, the end-of-file can be generated by pressing 'ctrl-D' (in Cygwin UNIX), Albert isn't sure if it is the same in Windows.

Keyboard input (via stdin) is typically line-buffered.

C reads line by line, therefore our C program may not set the input until the user enters the newline character.

the input is stored into a buffer, which is not submitted to the C program, when we enter a newline character, the buffer is submitted to the C program.

The following loop can be used to copy a file using I/O redirection:

```
#include <std.io>

int main(void) {
    int c;
    while ((c=getchar()) != EOF)
        putchar(c);
    return 0;
}
```

If we save this code into a 'file.c' and run from the command line, we can keep on typing until we hit enter.

This loop is useful.

Standard Idiom to process stdin Character-by-Character

```
int c;
while((c = getchar()) != EOF)        /* process c */
```

Example 1) "Converting" a file to all uppercase

```
int c;
while((c =getchar()) != EOF)
    putchar(touppoer(c));
return 0;
```

Example 2) Counting the number of lines in the file

```
#include <stio.h>

int main(void) {
    int c;
    size_t nlines = 0;

    while ((c = getchar()) != EOF)
        if(c == '\n')
            nlines++;
    printf("%lu\n", (unsignedlong)nlines);

    return 0;
}
```

NOTE: count the number of newline characters. This won't work correctly if last line does not have a newline character

Here is the bash command:

```
$./a.out < copy.c
```

NOTE: copy.c contains text with many lines

Reading line-by-line and fgets()

This should be used for interactive user input

```
fgets(buffer, buffsize, stream)
```

This function extracts characters from the *stream* and stores them into the *buffer* until one of three things happens:

NOTE: in all 3 cases a NULL '\0' character is appended to the buffer

1) end-of-file becomes true

2) the newline character has been read and stored

3) buffsize --> (-1) characters have been read and stored

fgets() returns the *null pointer* on a read error or end of file with no characters read.

The null pointer can be denoted by NULL or simply by 0

0 has many different meanings, it can mean the number 0, the character 0, or the null pointer

```
char line [100];
while(fgets(line, 100, stdin) != 0) /* while not false */
    printf("%s", line);
```

NOTE: '!=0' is a double negative, we can just say while TRUE instead of saying while NOT False]()

Standard Idiom to Process a Stream Line-by-Line

```
#define LINESIZE        /* a large integer value */
char line[LINESIZE];
while (fgets(line, LINESIZE, stream))       /* while true*/
    /* process line */
```

NOTE: This while loop is read as "As long as we read a line, process the line"**

Explanation: If you specify a line size that is too small, and call fgets() with a stream that is too big for the buffer, you will get some unprocessed characters that are still inside the input buffer. Another subsequent call of fgets() will not wait for user input and will process the unprocessed characters in the buffer from its last call. The lesson here is to use a large buffer.

Reading Item-by-Item using scanf(), fscanf(), sscanf()

scanf - reads from stdin

fscanf - can specify which stream to read from

sscanf - can specify which string to read from

Example:

Never use scanf for interactive input!! --> use fgets and sscanf instead so we will never get into an infinite loop. Scanf can never throw away invalid inputs, so it will infinitely loop if you give valid inputs.

```
int n;
scanf("%d", &n);        /* need to check the return value of scanf */
```

NOTE: Remember that everything is passed by value into functions, we are passing 'copies' into functions. Function scanf gets a copy of n, not the actual n, therefore we need to add an ampersand in front of n to specify that we want the copy of the address of n, which is just as good as the address of n. The '&' is the 'address-of' operator.

The definition of a pointer is a variable that stores the address. Whenever we put an '&' in front of a variable, we are asking for the address of the variable in memory.

fscanf: basically the same as scanf, except that it takes an extra argument (the 1st argument) that specifies the *stream* to scan from sscanf: basically the same as scanf, except it takes an extra argument (the 1st argument) that specifies the *string* to scan from

%d: integer in base 10

%f: floating point

%lf: double

%c: character

%s: word (s is for string, but it will read the first word, not the whole line, because scanf skips leading whitespaces)

scanf typically skips leading whitespace

```
int a = 10, b = 5, c = 7;

/* Example 1 */
a = sscanf("  123abc", "%d", &b);
/* a = 1, b = 123 */

/* Example 2 */
a = sscanf("  123 456", "%d%d", &b, &c);
/* a = 2, b = 123, c =  456*/

/* Example 3 */
a = sscanf("  hello 123", "%d%d", &b, &c);
/* a = 0, b is unchanged, c is unchanged */

/* Example 4 */
a = scanf("  ", "&d", &b);
/* a = EOF, b is unchanged */
```

NOTE: in example 1, the scan stops at the 3, because it cannot scan the 'a', because we specified "&d"

NOTE: in example 4, there is nothing to scan, because we specified an int with "%d" and there are only spaces, therefore a is EOF

```
float f = 1.23; /* default value to avoid random value */
char s[128];

/* Example 4 */
a = sscanf(" 12.345hello", "%d%f, &b, &f");
/* a = 2, b = 12, f = 0.345 */

/* Example 5 */
a = scanf("123hello world", "%d%s", &b, s);
/* a = 2, b = 123, s = "hello"
```

NOTE: array name can already be used as an address, don't need a & (name of array is starting position of array)

```
example scanf("%127s", s)
    /* read at most 127 characters & pad with null characters */
```

I/O can be thought of as conversion between data types & text example: datatype -> text

output is printf and input is scanf

Example: IMPORTANT how to use user directed input --> summing integers obtained from the user interactively

```
int LINESIZE =  1024;       /* define to prevent overflow error */
int n, sum = 0, char line[LINESIZE];

while (1) {
    printf ("Enter an integer:");
    if (!fgets(line, LINESIZE, stdin)) {
        clearerr (stdin);
        break;
    }
    if (sscanf(line, "%d", &n) == 1)
        sum += n;
    printf("%d\n", sum);
}
```

123hello --> scan for %d %c

fscans --> 123

Standard input is typically line buffered; if you don't print a new line character, the message may not show immediately

standard error is not buffered. For debugging, print to standard error! (or always remember to print a new line)

File I/O

reminder: I/O performed through streams

3 steps

1) open a file

2) perform I/O

3) close the file

Opening a file: this associates a stream with the file

In C, a stream has type FILE

Standard idiom to open a file

```
FILE *fp;           /* white space condensed, fp name of variable */

if ((fp = fopen(filename, mode)) == 0) {        /* open fails */
    perror("fopen");
        /* additional error-handling eg. exit(1) */
}
```

Both FILE *fp and FILE*fp are OK but FILE *fp is recommended

ex. FILE *fp, *fp; --> 2 file pointers, note that * is repeated

FILE *fp1, fp2 --> fpl is a file pointer, fp2 is a FILE!!

perror is used to print a system error message. It only works if a function in the standard C library fails, it sets *errno*

errno is a global integer variable that contain library functions used to store an error number when the fail perror looks at errno & prints a corresponding error message

when calling perror, we should pass in the name of the function that may fail

October 4, 2018

When a file is open for read/write, generally we need to call fseek if we switch from read to write or from write to read

eg. changing a file to all uppercase character by character

```
fp - FILE*

int c;
while ((c = fgets(fp))!=EOF) {
    fseek(fp, -1, SEEK_CUR);
    fputc(toupper(c), fp);
    fseek(fp, 0, SEEK_CUR);
}
```

file content: _hello --> beginning

h_ello --> after fgetc

_hello --> after fseek(fp, -1, SEEK_CUR)

H_ello --> after fputc('H', fp)

There are 3 indicators associated with a stream:

1) file position indicator: indicates our current location in the stream (ftell, fseek)

2) end-of-file indicator: specifies whether the end-of-file condition is true (can be tested using feof)

3) error indicator: set if there is an error (can be tested using ferror)

Pointers

A pointer is a variable that stores an address.

int n = 1;

'int n' declares an integer variable

'= 1' stores an integer in it

We need to be able to:

1) Declare a pointer

2) get addresses

3) go to the destination stored specified by the address stored in a pointer

Sources of Addresses

1) names of arrays

2) the address-of operator

int n = 1;   &n - address of n

int a[10];   a - can be used as the starting address of the array

Going to the destination specified by an address or pointer (dereference)

p - pointer or address

*p is the destination specified by the pointer or address

*p is the dereference pointer

think of this as saying: go to the destination

```
int a[] = {3, 2, 7};
printf("%d," *a);       /* 3 */
```

NOTE: the starting address of an array is the address of its first element

a == &a[0] is true

Declaring a pointer

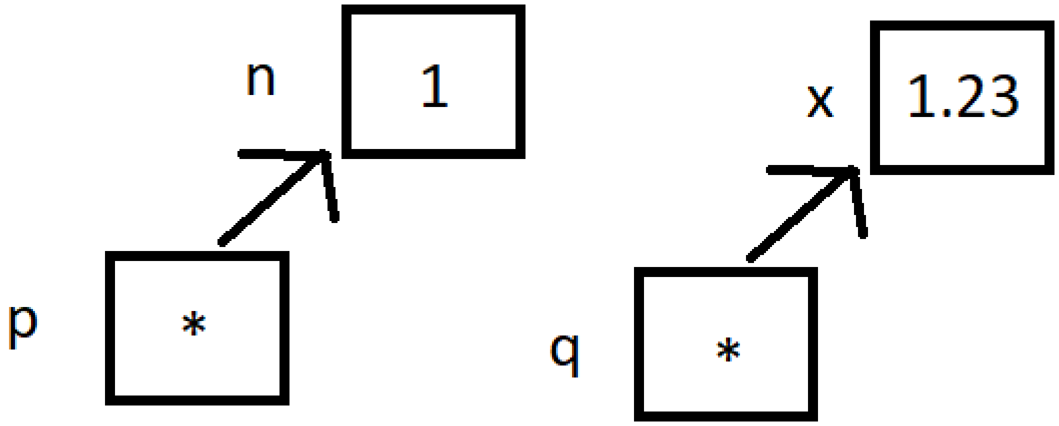
Examples:

1) variable to store the address of an int

```
int p;          /* p is a pointer to an int */
                /* read from right to left */
                /* is read as "pointer to" */

int n = 1;
p = &n;                 /* OK, storing the address of an int in p */
printf("%d", p);        /* 1 */
*p = 2;                 /* changes the value of n to 2 */
```

2) variable to store the address of a double

```
double x = 1.23;
double *q = &x;
*q = 5.67;              /* changes x to 5.67 */)
```



We can think of the dereference operator as saying: follow the arrow

Definition: We say that the pointer p points to the object x if p contains the address of x

Example:

```
int *a;
```

int --> destination type: this is needed when we try to go to the destination (i.e. dereference the pointer)

October 10, 2018

There are 3 indicators associated with a stream:

1) file position indicator: indicates our current location in the stream

2) the End-Of-File indicator: specifies whether the end-of-file condition is true

can be tested using feof() function that returns true or false

3) error indicator: set if there is an error

```
/* fp is a file pointer */

clearerr(fp);
```

clears the end-of-file indicator and the error indicator. If the error indicator is set, all later I/O operations are not guaranteed to work

Pointer review:

A pointer is a variable that stores an address. When you see a declaration, you can read it right to left, 'p is a pointer to 'int'

```
int n = 123;
int *p = &n;
printf("%d", *p);       /* *p is the dereference operator */
```

123 is the dereference operator

the dereference operator basically means follow the arrow

DEFINITION: we say that x points to y if x contains the address of y

Examples 1)

```
int *p=  &n;
```

what is the type of p? It is a pointer to n, written and represented as: int*

what is the type of *p? It is an int

Example 2)

```
int *p;
```

there are two ways to 'read' this statement

1. *p is an int
2. p is an int

Example 3)

```
int **q
```

there are three ways to 'read' this statement

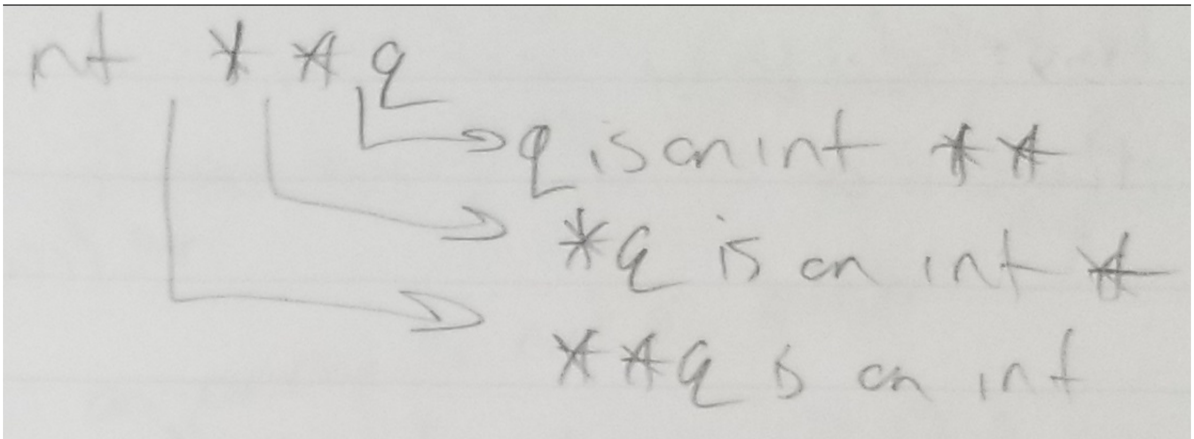
q is an int**

*q is an int *

** is an int

```
int n = 123;
int *p = &n;
int **q = &p;
**q = 456;
```

*NOTE: q is a pointer to a pointer (double pointer) to an int. It points to a pointer. That is why you need two *'s*

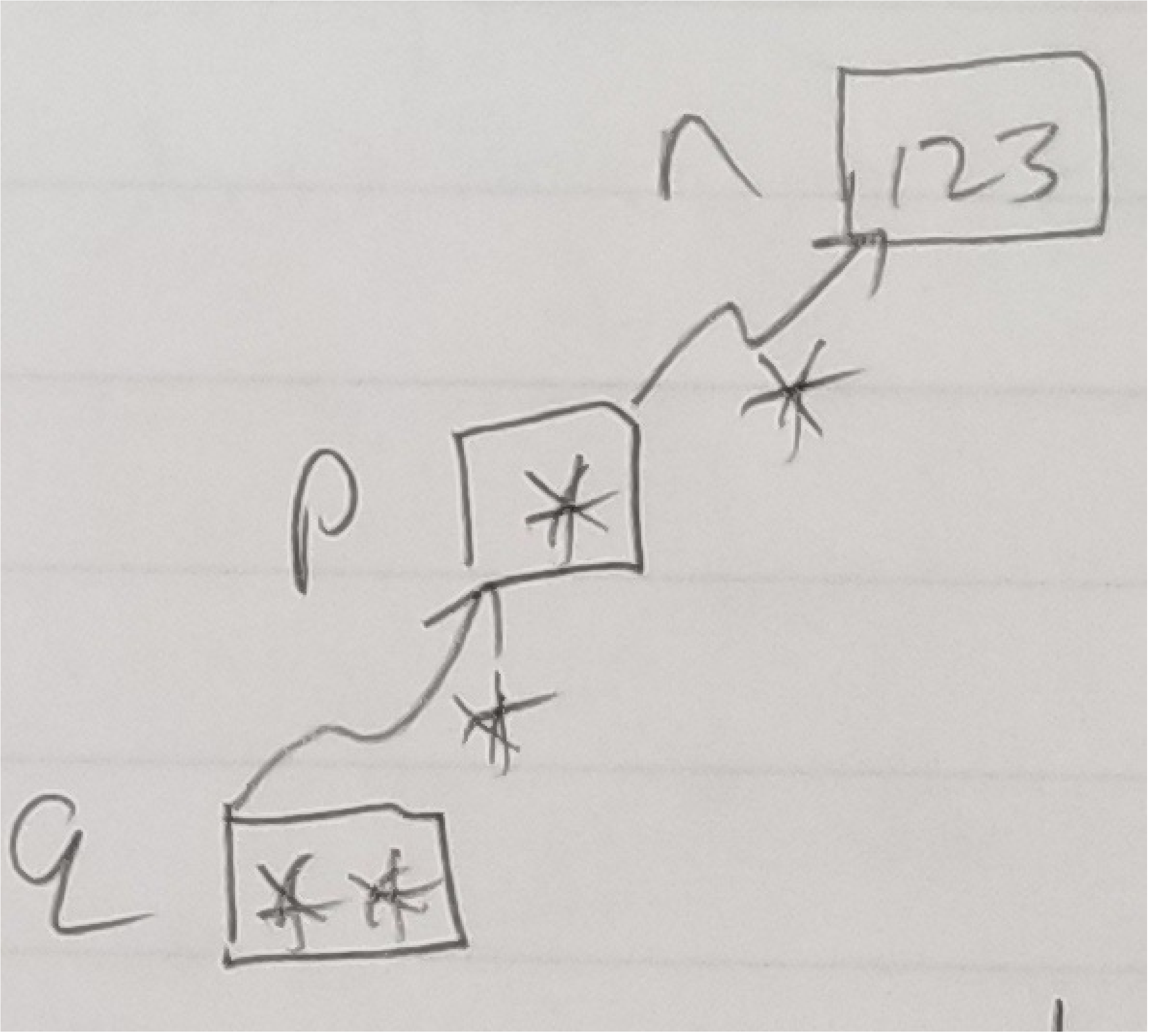


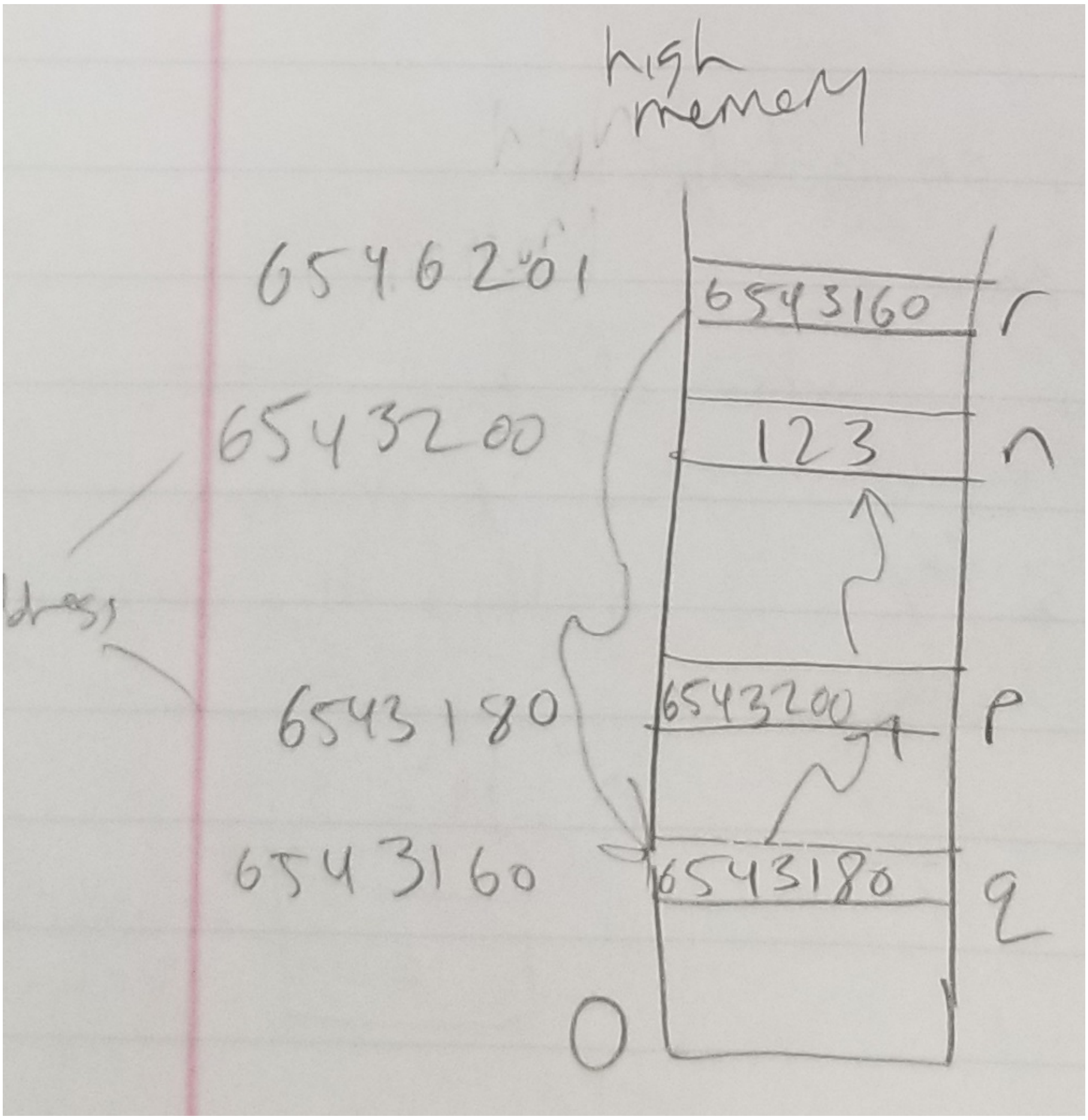
nt * * q

q is an int **

*q is an int *

**q is an int


n
123
*
p
*
*
q
**



```
int n = 123; /* n's address is 6543200 */
int *p = &n; /* p's address is 6543180 */
int **q = &p; /* q's address is 6543160 */
int ***r = &q; /* r's address is 6543140 */
printf("&d", &p); /* 6543180 */
printf("&d", &q); /* 6543180 */
printf("&d", &r); /* 6543180 */
```

NOTE: assume a 32-bit system (4-byte addresses) and an int is 4 bytes

*r is q (r contains the address of q, *r is an alias for q)

*q is p (think of *q as an alias for p)

*p is n (think of *p as an alias for n)

pointers give us an indirect way to get at an object

```
int *p = &n;
```

this does not assign to *p, it assigns &n to p

```
int *p
/* is interchangeable with */
int * p

/* so */

int*p = &n;
/* is the same thing as */
int *p;
p = &n;
```

Further elaboration:

```
#include <stdio.h>

int main(void) {
    int n = 123;
    int *p = &n;
    int **q = &p;
    int ***r = &q;

    printf("n:: %p: %d\n", &n, n);
    printf("p:: %p: %p: %p\n", &p, p, *p);
        /*the content of p is the address of n */
    printf("q:: %p: %p: %p\n", &q, q, *q);
        /* the content of q is the address of p */
    printf("r:: %p: %p: %p\n", &r, r, *r);
        /* the content of q is the address of p */

    return 0;
}
```

Pass by Reference

Everything in C is pass by value

How to have the effect of pass by reference, as in how to enable a function to change a non-global object outside of it.

Three steps to pass by reference instead of passing by value:

1) add an extra star (*) in front of the parameter you want the function to change

2) within the function, dereference (put a star in front of) that parameter to get the original object

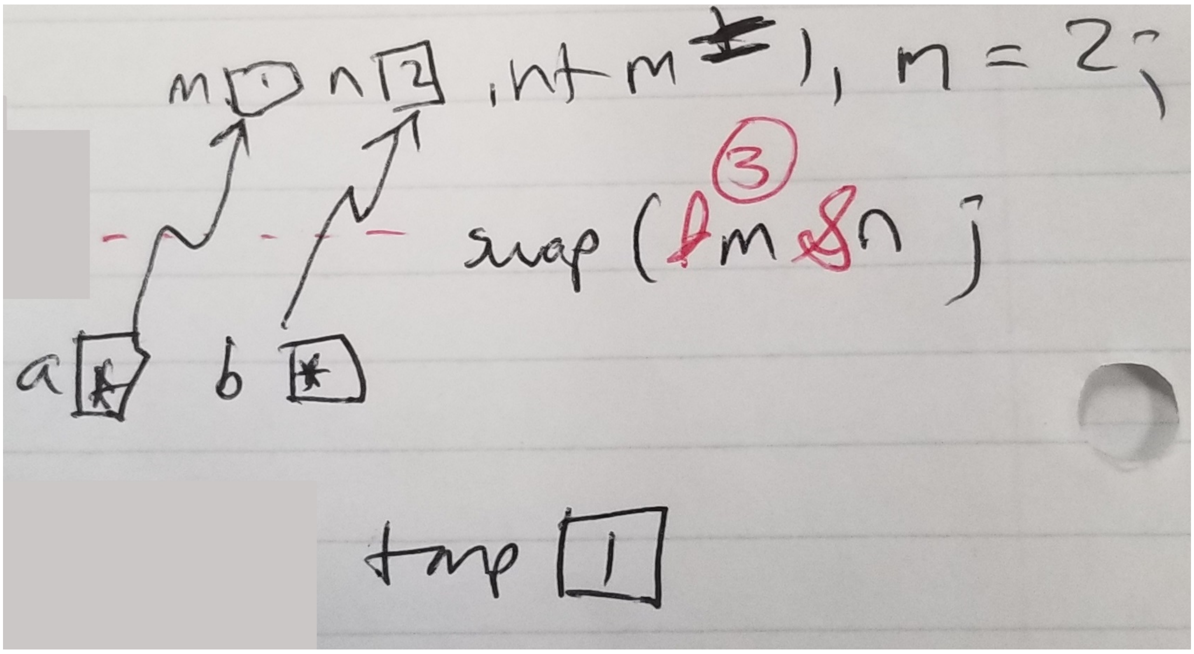
3) when calling the function, pass in the address of the object you want the function to change

Examples 1) function to triple an int

```
void triple(int *n) {             /* Step 1 */
    *n *= 3;                      /* Step 2 */

int main(void){
    int x = 1;
    triple(&x);                   /* Step 3 */
    return 0;
}

    /* alternately */

int main(void) {
    int *p = &x;
    triple(p);                /* Step 3 */
    return 0;
}
```

Example 2) function to swap two integers

```
void swap(int *a, int *b) {      /* Step 1 */
    int temp = *a;               /* Step 2 */
    *a = *b;
    *b = temp;
}

int main(void) {
    int m = 1;
    int n = 2;
    swap(&m, &n);                /* Step 3 */
    return 0;
}
```



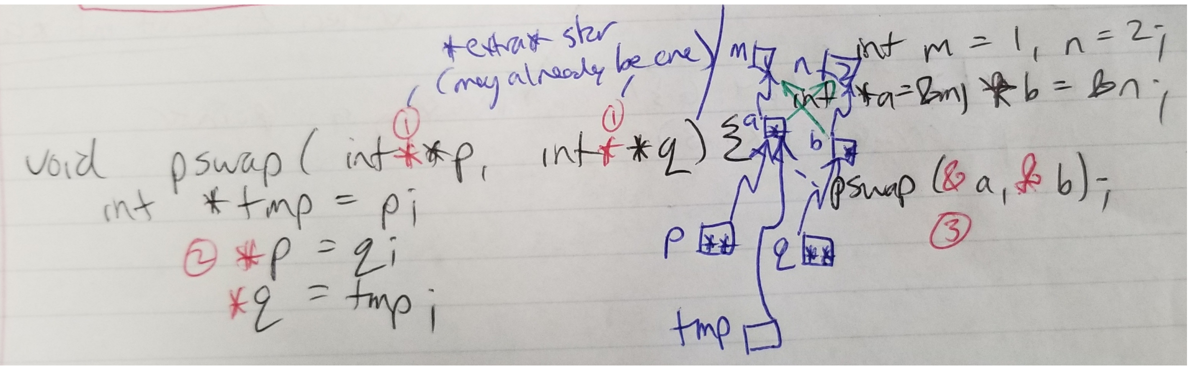
IMPORTANT POINT given 2 pointers p & q of the same type

p = q makes p point to the same thing as q, now pointing to the same address, p's point

making two pointers equal to each other makes them point to the same thing. They share the same basic address

EXAMPLE 3) function to swap 2 pointers to int

```
void pswap(int **p, int **q){   /* Step 1 */
    int *tmp = *p;              /* Step 2 */
    *p = *q;
    *q = tmp;
}

int main(void){
    int m = 1, n = 2;
    int *a = &m, *b = &n;
    pswap(&a, &b);              /* Step 3 */
    return 0;
}
```



EXPLANATION: what is inside the function? p and q. They are double pointers (their boxes have two stars in them). What is outside the function? a and b, m and n. m and n have boxes holding 1 and 2 respectively. a and b are pointers (they have boxes holding a star that points to m and n respectively). temp contains the same content as a (temp's box now has a star that points to m's box).

Pointers, Addresses, and Arrays

Pointers store an address

Array names can be used as addresses

```
int a[] = {2, 4, 6,}
int *p = a                  /* ok */
```

We can index into the array using p:

```
p[0], p[1], p[2]

/* same as */

a[0], a[1], a[2]
```

NOTE: we have been doing this all along with regards to arrays

```
int arr_sum(const int a[], size_t n){       /*a is a pointer*/
    int sum = 0;
    size_t = i;
    for(i = 0; i < n; i++){
        sum += a[i];                        /*indexing a pointer*/
    }
    return sum;
}

int main(void){
    int x[] = {2, 4 ,6};
    int y = arr_sum(x, 3);
    return 0;
}
```

*NOTE: what is type of a? It is actually a pointer (as it stores the starting address of an array).   It's type can be written as const int *a*

October 11, 2018

A pointer to T (some type) can point to an array of T's; it simply meant the pointer is sorting to the 1st element of the array

We can use the index notation with the points as well as with arrays

As a parameter of a function, T a[] ≡ T *a

T is a type, ≡ means equivalent to

NOTE: an array is not a pointer in general. The whole point of pointers is to store an address. An array name is the address of the first element.

pointers and arrays are different objects

```
int a[] = {3, 2, 1};
int *p;
p = a;          /* Ok, points to first element of array */
a = p;          /* does not compile; cannot assign to an array */
```

The difference between p and a is like the difference between int n and 123

Think of the array name as a fixed number, it's address

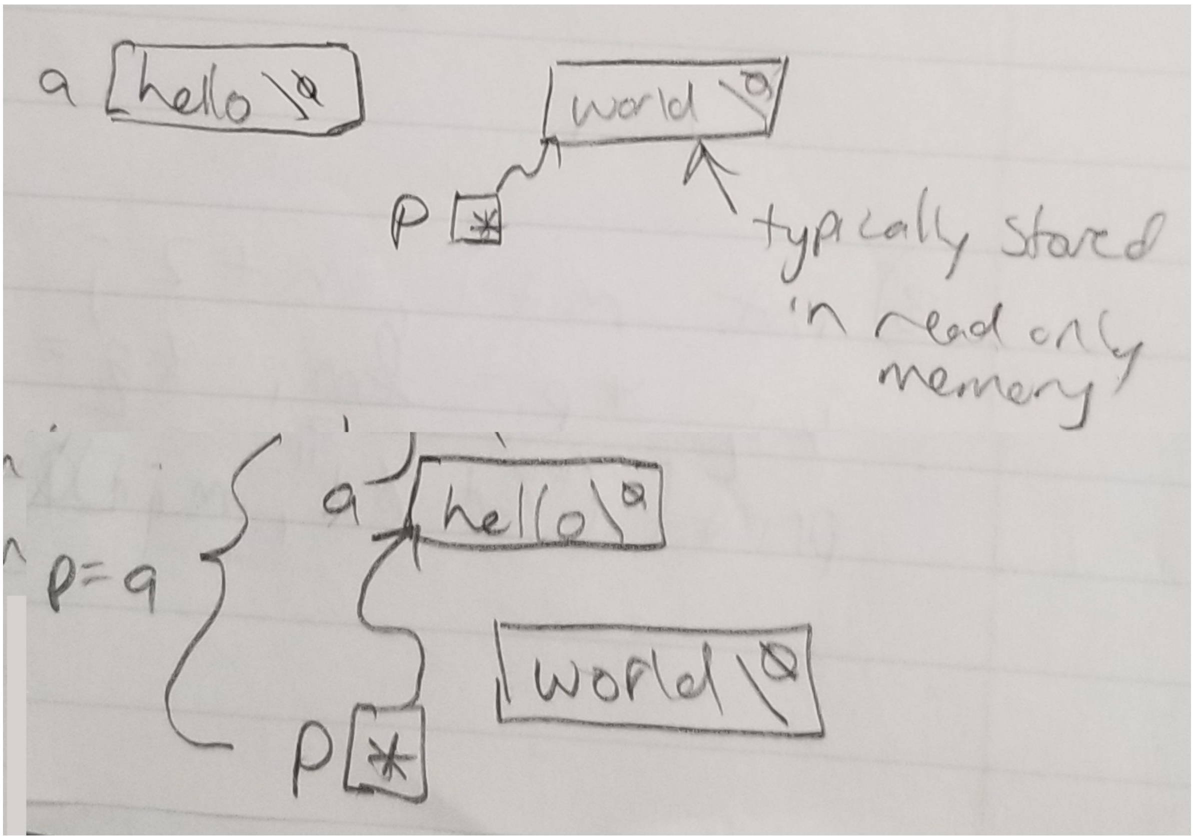
```
int n;
n = 123;        /* Ok */
123 = n;        /*does not compile */
```

Examples 1)

```
char a[] = "hello";
char *p = "world";
a = p;              /* doesn't compile */
p = a;
```

doesn't compile because it can't assign an array to an address

EXPLANATION: Think of 'a' as a number like 123. You can never assign to 123. Now, p is a pointer, and its purpose is to store an address.



P has a box with a star in it. It points to a box that has world\0 in it. But there is no reference. After p = a; P's box has a star that points to A's box.

Example 2)

```
char a[] = "hello";
char *p = "world";           /* See 1. below */
a = p;                       /* See 2. below*/
p = a;
a[0] = p[0];                 /*OK*/
p[0] = a[0];                 /* See 3. below */
printf("%s", a);             /*wello*/
printf("%c", *p);            /*w*/

int a[] = {3, 2, 1};
int *p = a;
printf("%d", *p);            /*3*/
```

1. we didnt allocate memory to store string, it is a string constant

2. can't compile --> can't assign to an array name

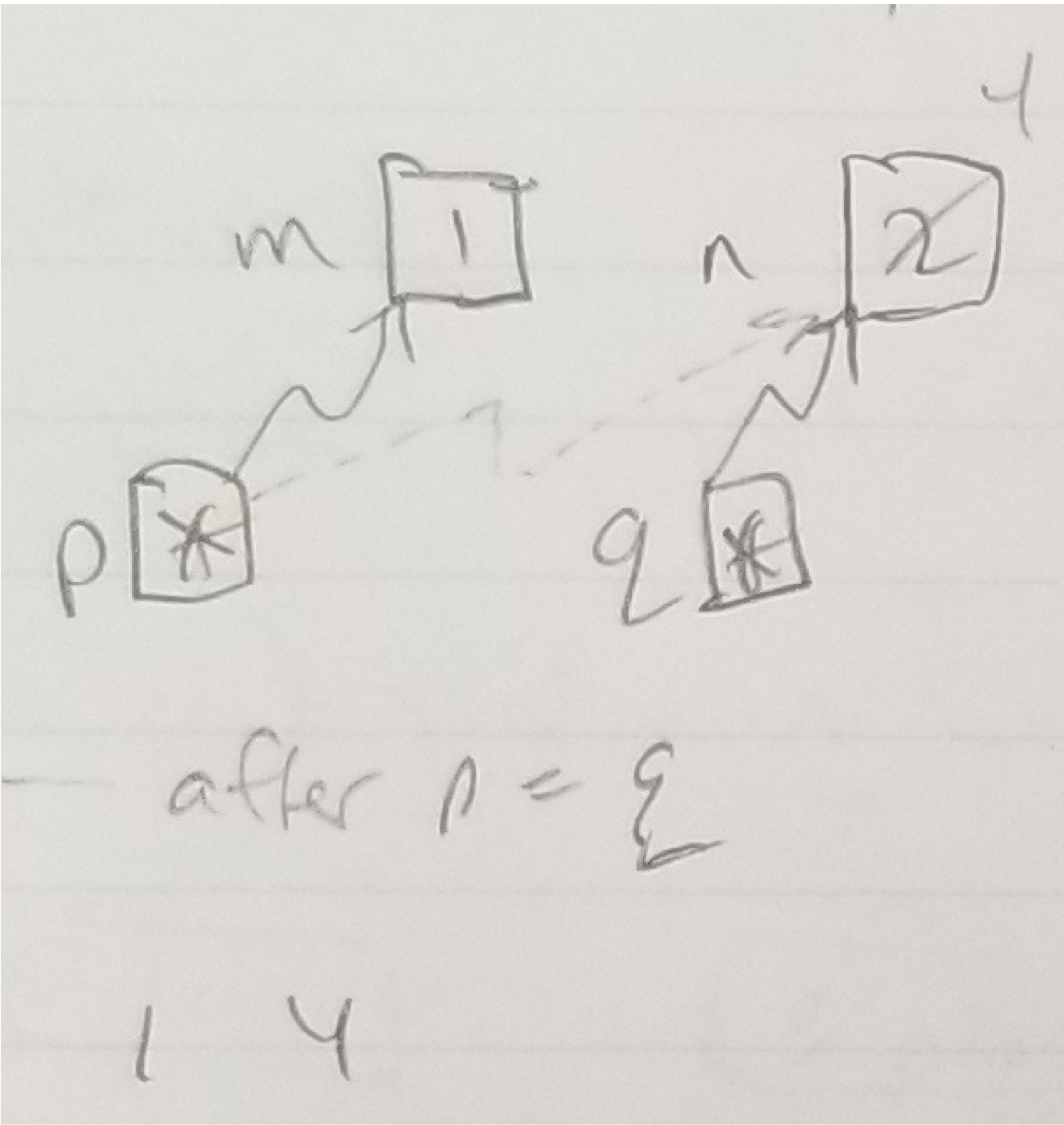
3. invalid, we can't change a string constant

NOTE: a string constant is stored in read-only memory

Example on Pointers 1)

```
int m = 1, n = 2;
int *p = &m, *q = &n;
p = q;
*p += 2;
printf("%d %d", m, n);           /*1 4*/
```

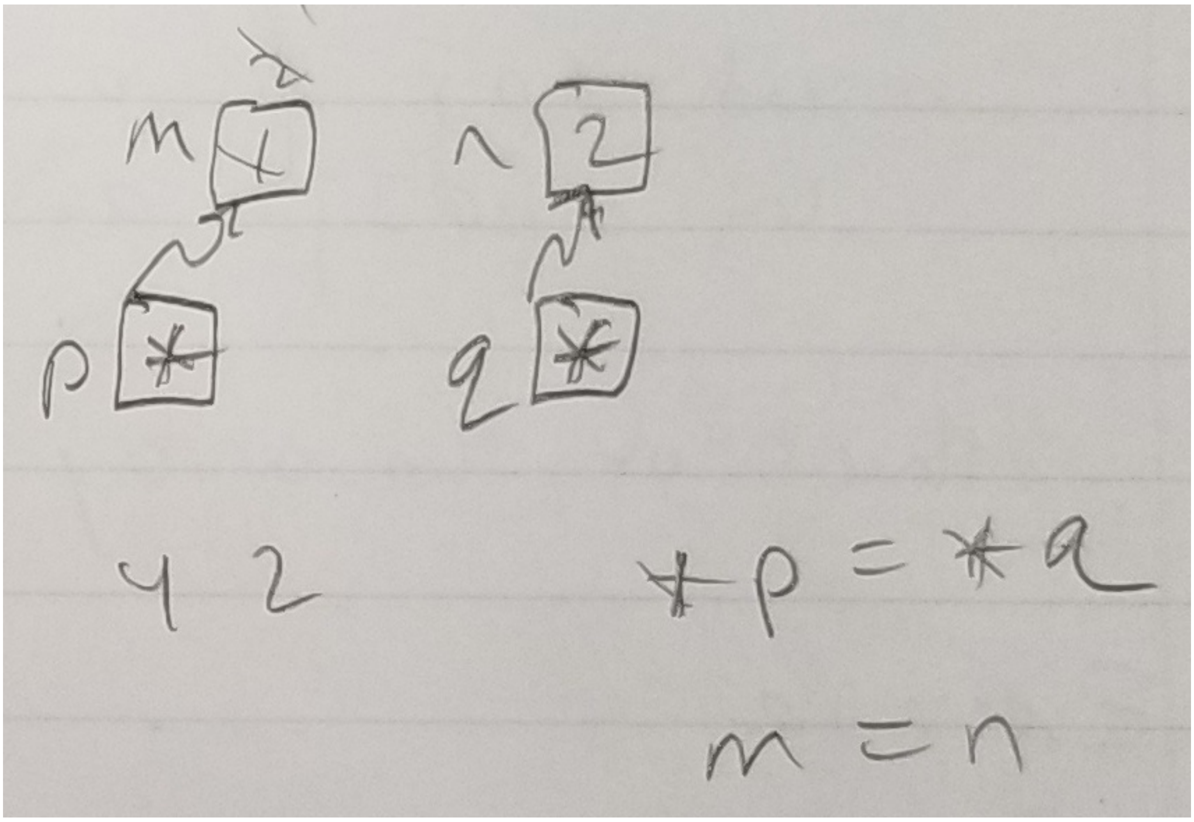
both pointers now point to the same thing:



Example on Pointers 2)

```
int m = 1, n = 2;
int *p = &m, *q = &n;
*p = *q;
*p += 2;
printf("%d %d", m, n);          /*4 2*/
```

*p is m, *q is n --> *p = *q is just like saying m= n;

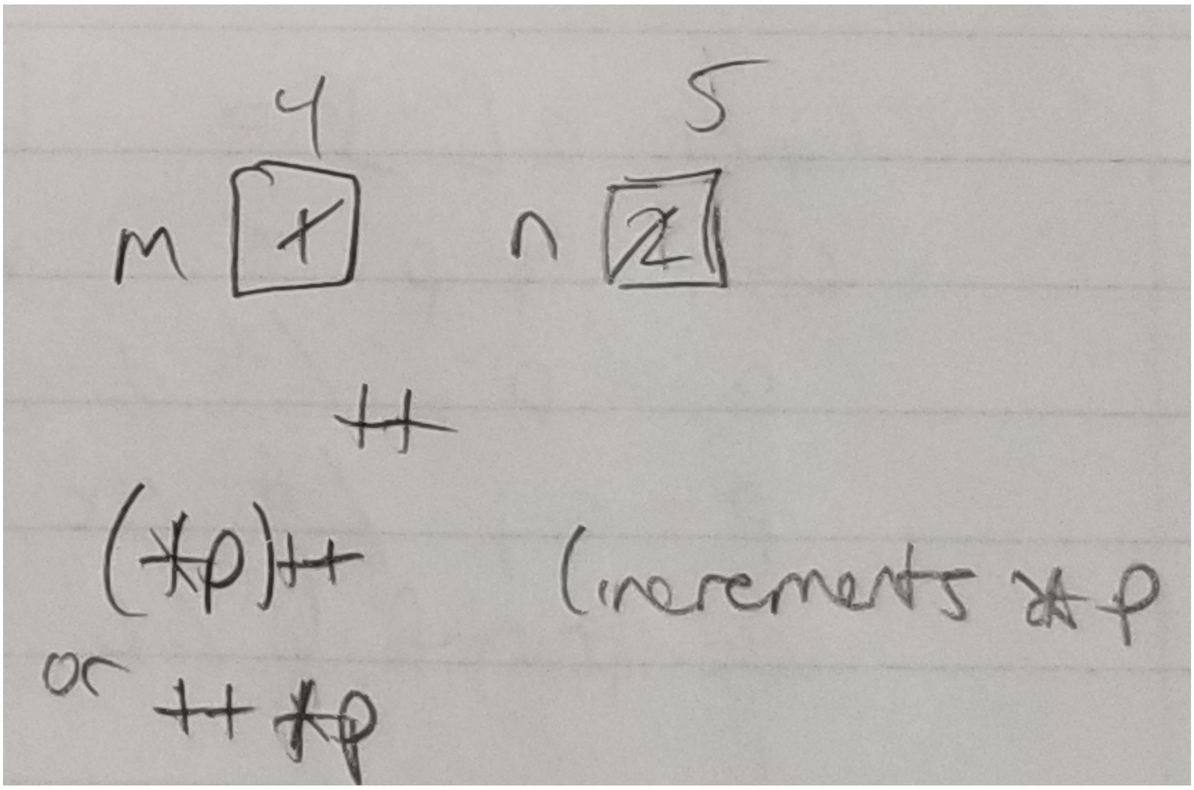


Example on Pointers 3)

```
int m = 1, n = 2;
int *p = &m, *q = &n;
*p = *q + 2;                    /* m = n + 2 */
*q = *p + 1;                    /* n = m + 1 */
printf("%d %d", m, n);          /*4 5*/
(*p)++;
++*p;
```

NOTE: brackets needed incrementer/decrementer is trailing pointer, but unnecessary if preceding it

NOTE: if ever unclear, draw a picture



Primitive Types:

Integral Types:

char / signed char / unsigned char

short (signed short) / unsigned short

int (signed) / unsigned int (unsigned)

long (long int) / unsigned long (unsigned long int)

Floating Point Types:

float

double

long double

ANSI C does not specify the exact size of each type. There is a size of operator that can be used to find out the size of each type (or variable)

IMPORTANT: by definition, size of char is 1 byte, and everything else is relative to char

What is the type of:

1	int
1l / 1L	long
1u / 1U	unsigned double
1.0	double
1.0f / 1.0	float

What happens when we mix operands of different types in a binary operation?

1 (int) + 1.2 (double) --> a double because int 'promoted' to double to carry out operation

1U (unsigned) - 2

October 17, 2018

Automatic conversions:

long double **<--** double **<--** float **<--** unsigned long **<--** long **<--** unsigned int **<--** int

anything that is lower we convert to that which is higher

Unsigned type is higher than signed type

If we have 2 operands of different type in a binary operation, the on that is lower in this table is converted to the type of the one that is higher

(int) 1 - 2U (unsigned int)

1 is converted to unsigned int and subtraction of unsigned are performed

1 - 2U < 0 is false

NOTE: an unsigned can never be less than zero

```
int i;
for (i = 0; i < strlen(i); i++)     /* */
```

may generate compiler warning --> comparison between signed and unsigned type

i is signed, but strlen is unsigned types

It is dangerous to mix signed and unsigned types

why there are no unsigned types in java

Precedence and Associativity

They determine how terms are grouped

1 + 2 * 3 is the same as (1+2) * 3 or 1 + (2 * 3)?

1 + 2 * 3 ≡ 1 + (2 * 3) because * has higher precedence than +

1 - 2 + 3

is this (1 - 2) + 3 or 1 - (2 + 3)

(1 - 2) + 3 ≡ (1 - 2) + 3

+ and - are on the same precedence level: they associate left to right

```
(c = getchar()) != EOF
c = getchar() != EOF ≡ c = (getchar() != EOF)
                       != has higher precedence than =
```

Order of Presentation:

In many cases, C does not specify the order of evaluation

```
f(a,b)
```

is a evaluated before b or vice versa??

```
f(g(), h())
```

g and h both definitely need to be evaluated before f, but which is evaluated first?

```
int n = 1;
int x = n++ + n++;
```

value of x is unspecified!!

compiler will generate a warning

IMPORTANT: We should not modify the same variable more than once within an expression

However, there are situations where the order of evaluation is specified

|| and &&

Example 1:

```
int a = 1, b = 0, c = 2;
c = --a && ++b;
/* a: 0 b: 0 c: 0 */
```

a = 0, therefore, b is not evaluated, therefore c is zero

Example 2:

```
int a = 1, b = 0, c = 2;
c = a-- && ++b;
/* a: 0 b: 1 c: 1 */
```

a = 1, ++b = 1

Example 3:

```
int a = 1, b = 0, c = 2;
c =  --a || ++b
/* a: 0 b: 1 c: 1 */
```

a = 0, and b = 1

Example 4:

```
int a = 1, b = 0, c = 2;
c = a-- || ++b;
/* a: 0 b: 1 c: 1 */
```

a = 1, b not evaluated

Control Flow

1) if/else if/else

```
if (argc == 1) {
    /* relevant code here */
} else if
        /* relevant code here */
} else {
            /* relevant code here */
}
```

2) switch case

```
switch (choice) {
    case 'y': case 'Y':
        /* relevant code here */
        break;
    case 'n': case 'N':
        /* relevant code here */
        break;
    default:
        /* relevant code here */
}
```

'choice' must be some integer type

3) while loop

IMPORTANT: n++ evaluates n then adds, ++n adds first, then evaluates

Example 1:

```
int n = 2;
while (++n < 7)
    printf("%%d", n);
/* loops through 4 times */
/* prints "3456" */
```

Example 2:

```
int n = 2;
while (n++ < 7)
    printf("%%d", n);
/* loops through 5 times */
/* prints "34567" */
```

Example 3:

```
int n = 2;
while (n < 7)
    printf("%%d", ++n);
/* loops through 5 times */
/* prints "34567" */
```

Example 4:

```
int n = 2;
while (n < 7)
    printf("%%d", n++);
/* loops through 5 times */
/* prints "23456" */
```

4) do/while loop

```
do {
    /* relevant code here */
} while (/* relevent condition */);
```

5) for loop

for (initialization; test to continue in loop; updating)

```
int is_palendrome(const char s[]) {
    size_t i, j;
    for (i=0, j=strlen(s); i<j; i++, j--);
        if (s[i] != s[j-1])
            return 0;
        return 1;
}
```

Standard idiom to process an array backwards

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

char s[] = 'hello';

for(i = 5 i > 0, i--)
    s[i] = toupper(s[i]);

size_t i;

for i = N; i > 0; i--)
    /* process a[i-1] */
```

Standard idiom to process a string backwards

```
s = string

size_t i;
for(i = strlen(s); i > 0; i--)
    /* process s[i-1] */
```

6) break / continue

must be used within loops (except break in switch statements)

eg.

```
while ( ... ) {
    if ( ... )
        break;
    if ( ... )
        continue;
    if ( ... )
        break;
}
```

continue makes flow of control skip to the end of the while loop, which then loops back up to the beginning

```
i = 0;
while ( ... ) {
    /* what if you have a a continue in here? */
    i++;
}
```

IMPORTANT: the loop never increments i, which can't happen with a for loop

7) goto

e.g.

```
goto error;     /* must be written within the same function */

error:      /* label */
```

8) Ternary operator --> a ? b : c

Bit Manipulation

~ >> << & | ^ and their variants like &= |= etc

~ (complement) --> flips 0's and 1's

Assume 16 bit unsigned ints

```
unsigned int n = 0x13ad;        /* hexadecimal */
```

REMINDER: 0x is a prefix designation denoting hexadecimal format

n: 0x13ad = 0010 0111 0101 1101

~n = 1110 1100 0101 0010 = 0xec52

<< (left shift): always shifts in 0's

```
unsigned int n  = 0x13ad;
```

n: 0x13ad = 0010 0111 0101 1101

cut off first 3 and add three zeros to the right

n << 3: 1001 1101 0110 1000 = 0x9268

>> (right shift):

for unsigned types, always shift in 0's

for signed type, if value is non-negative, shift in 0's

for signed type, if value is negative, may shift in 0's or 1's

n: 0x13ad = 0010 0111 0101 1101

n >> 3 = 0000 0010 0111 0101 = 0x0275

& (and):

1 if both bits are 1; 0 if otherwise

| (or):

0 if both bits are 0; 1otherwise

^ (xor):

0 if bits are the same; 1 otherwise

Examples:

unsigned a = 0x13ad, b = 0xf1c2; 0xdeadbeef

a = 0x13ad = 0001 0011 1010 1101

b = 0xfc2 =   1111 0001 1100 0010

a|b = 1111 0011 1110 1111 = 0xf3ef

a&b = 0001 0001 1000 0000 = 0x1181

a^b = 1110 0010 0110 1111 = 0xe26f

October 18, 2018

REVIEW:

3)

```


int main (int argc, char *argv[]) {
    if (argc != 2) {
         ------
        return 1;
    }
    if ((fp = fopen(argv[1], "rb")) == 0) {
        perror("fopen");
        return 2;
    }
    while ((c = fgetc(fp)) != EOF)
        if (isalpha(c))
            nalphas++
            printf("%lu\n", (unsigned long) nalphas);
    if(fclose(fp) !- 0) {
        perror("flose");
        return 3;
    }
}

int main(argc, char *argv[]) {
    int c, count;
    if (argc != 2)
        return 1;
    fopen(arg[1], "r")
    while ((c = fgetc(argv[1]))
        if(isalpha(c))
            count++;
    printf("%d", count);
    return 0;
}
```

2) (b)

```
int all_digits(const char s[]) {
    size_t i;
    for (i = 0; s[i] != '\0'; i++)
        if (!isdigit(s[i]))
            return 0;
    return 1;
}
```

c)

```
void lowercase_copy(char dest[], count char src[]) {
    size_t i;
    for (i = 0; src[i] != '\0'; i++)
        dest[i] = tolower(src[i]);
    dest[i] = '\0';         /* or: dest[i] = src[i]; */
}
```

d)

```
void reverse_copy(char dest[], const char src[]) {
    size_t i, j;
    for (i =  strlen(src), j = 0; i > 0; i--, j++)
        dest[j] = src[i - 1];
    dest[j] = '\0';
}
```

IMPORTANT: don't ever use strlen -1

e)

```
void ltrm_copy(char dest[], const char src[]) {
    size_t i;
    for (i = 0; src[i] != '\0;' i++)
        if(!isspace(src[i]))
            break;
    for(j = 0; src[i] != '\0'; i++, j++)
        dest[j] = src[i];
    dest[j] = '\0'
}
/*
void ltrm_copy(char dest[], const char src[]) {
    size_t i;
    for (i = 0; src[i] != '\0;' i++)
        if(!isspace(src[i]))
            break;
    for(j=o; src[i] != '\0'; i++, j++)
        dest[j] = src[i];
    dest[j] = '\0'
}*/
```

1.c)

```
void replace last(chars[], int oldc, int newc) {
    size_t i, pos = -1;
    for (i=0; s[i] != '\0'; i++)
        if (s[i] == oldc)
            pos = i;
    if (pos! = (size_t) - 1)
        s[os] = newc;
}
```

Questions:

what is : and how does it work?

both in the || and && and the switch

what the fuck is going on in the || and &&