Written by: Luyanda Mncube

-42

The hitchhiker's guide to staying alive at `Wethinkcode_` *South Africa*

Written by: Luyanda Mncube

"The answer to all things in Bootcamp is -42"

-e5r8p4

This is a notebook on the C Programming language I wrote whilst attending the `Wethinkcode_` *bootcamp in Johannesburg South Africa in preparation for daily exercises to help fellow bootcampers who want to avoid the dreaded "0/100" Fail screen. All improvements/suggestions welcome at:* `lmncube@student.wethinkcode.co.za`

Lesson 1

- C does not support object oriented programming
- C does not support information hiding (no support for polymorphism, encapsulation, and inheritance)
- C does not allow functions to be defined inside structures.
- C uses functions for input/output. For example `scanf` and `printf` .
  (FORBIDDEN by 42 Norm)
- C provides `malloc` and `calloc` functions for dynamic memory allocation, and `calloc` for memory de-allocation.
  (FORBIDDEN by 42 Norm… Before day06
- C does not provide String or Boolean data types. It supports primitive & built-in data types.
- C supports only Pointers (no references)

HOW TO COMPILE IN UNIX (using gcc)

Moulinette uses `-Wall -Wextra Werror` when compiling, and uses `gcc`. All warnings in the compiler basically turn to errors
So anything that even looks like it will crash will cause you to get a fail grade ☹

Example 1: Compile command

```
$> gcc -Wall -Wextra -Werror test.c
$> ./a.out
```

You can also compile multiple C files for the same program.

```
gcc -O0 -Wall -Wextra -Werror -ansi -o my_executable my_code.c
```

PRO TIP: Make an alias

To avoid having to type this long command at each compilation, make yourself an alias!
Open the ~ / SHELLrc file (replacing SHELL with sh, csh, tcsh, bash, 42sh, ...)
Add a line of type:
```
alias mycc 'gcc -O0 -Wall -Wextra -Werror -ansi'
```
Then type
```
source ~ / SHELLrc
```
And finally, to compile, do
```
mycc -o my_executable my_code.c
```

Lesson 2 – Variables & Character handling

Variables are little boxes that help us store stuff in programming.

To know the ASCII values of each letter, we use the ASCII table that can be accessed through `man ASCII`.

```
48 - 57 | '0' - '9'
65 - 90 | 'A' - 'Z'
97 - 122 | 'a' - 'z'
```

Example: The letter 'F' is set to ASCII 70 while the letter 'f' is ASCII 102. Other non-visible characters that may be useful:

```
00 | '\ 0' | NULL, 0                    //Usually at the end of strings, so
                                        our compiler knows when to stop
                                        reading a string

07 | '\ a' | bell, beep
08 | '\ b' | backspace
09 | '\ t' | tabulation
10 | '\ n' | Return to the line         //Skips to next line
11 | '\ v' | Vertical tabulation
13 | '\ r' | Carriage return            //Come back to the beginning of the
                                        Line
```

To display a letter, we will use this small function:

```c
#include      <unistd.h>

void    ft_putchar(char  c)
{
  write (1, &c, 1);
}
```

You can send a string to a function by using double quotes (" ") my string of characters \ n ".
Single quotes (') are used for single characters.

PRO TIP: Understanding NULL or '\0'

NB: NULL in c languages is for NOTHING. It is a macro, that has a specific meaning to the compiler. It is not the same as zero. If you return 0 in a function it has a meaning (the integer zero). NULL is typically more useful in things like arrays and memory allocation. (i.e. set all values of an array to NULL before use so that if one result is stored as '0' you will know a value went in that index, the value just happened to be zero)

Lesson 3 – Useful (And Useless) Functions

Write

- call to system
- uses the #include <unistd.h> directive
  Remember, numbers still fall on the asci table. So don't try convert, treat them the SAME as you would a normal char.

in the form
write(fd, buf, nbytes);

fd	It is the file descriptor which has been obtained from the call to open. It is an integer value. The values 0, 1, 2 can also be given, for standard input, standard output & standard error, respectively .
buf	It points to a character array, which can be used to store content obtained from the file pointed to by fd
nbytes	It specifies the number of bytes (essentially characters) to be written from the file into the character array.

To display words to screen

```
#include <unistd.h>

int main(void)
{
    if (write(1, "Hello World!\n", 13) != 14)
    {
        write(2, "There was an error writing to standard out\n", 44);
        return -1;
    }

    return 0;
}
```

PRO TIP: Some useless information

What is a descriptor? (Used for Manipulation of files)
In simple words, when you open a file, the operating system creates an entry to represent that file and store the information about that opened file. So if there are 100 files opened in your OS then there will be 100 entries in OS (somewhere in kernel). These entries are represented by integers like (...100, 101, 102....). This entry number is the file descriptor. So it is just an integer number that uniquely represents an opened file in operating system. If your process opens 10 files then your Process table will have 10 entries for file descriptors.

Printf

- Provides FORMATTED output
- Used with specifiers, use them when you know what type of variable you are outputting
- uses the #include <stdio.h> directive
- In the const char part, use \n if you need to output stuff after printing

```
in the form
#include <stdio.h>

int main()
{
    printf ("Characters: %c %c \n", 'a', 65);
    printf ("Decimals: %d %ld\n", 1977, 650000L);
    printf ("Width trick: %*d \n", 5, 10);
    printf ("%s \n", "A string");
}
```

A list of some specifiers

d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
f	Decimal floating point, lowercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed.	

Scanf

- The scanf function allows you to accept input from standard in, which for us is generally the keyboard.
- uses the #include <stdio.h> directive
- The scanf function uses the same placeholders as printf:

int uses **%d**
float uses **%f**
char uses **%c**
character strings (discussed later) use **%s**

Void type

- is a type, goes for BOTH function parameters & function returns
- So, a void pointer in c is one without a type, but this actually doesn't matter (in the example, a VP function with a VP variable is created to act as a place holder to convert a char to an int)
- Therefore, void pointers are POLYMORPHIC
- In c++, when void is used as a parameter, it's basically casting a variable (i.e void void_one) but in c it's much simpler, just use void (i.e void) as a parameter

Ternary operators

- Different from Boolean operators because they have three sides instead of two.
- shortens and replaces need for if else
- if condition is true to left of question mark, returns first operand, if condition is false, returns second operand

In the form of:
 (condition ? return_true : return_false)

Example 1 – Shortening and if else statement using ternary operators:

int opening_time = (day == SUNDAY) ? 12 : 9;

Instead of:
int opening_time;

if (day == SUNDAY)
    opening_time = 12;
else
     opening_time = 9;

   Ternary operators can also be parameters for other functions (since ternary operators have specified output)

#include <stdio.h>

main()
{
    int a , b;
    a = 10;
    printf("Value of b is %d\n", (a == 1) ? 20 : 30);
    printf("Value of b is %d\n", (a == 10) ? 20 : 30);
}

Typedef (Useful for when we create classes in C)

   alias for a type name
   signified with _t (i.e. cards_t)
   Not much to it, just saves you time cause pointer definitions or even
   variables can take a while to type

Lesson 4 – Arrays

Arrays

- Little bookshelves that help you store information. More like badass variables
- Same as in C++, same logic and use
- Allocating and de-allocating memory are malloc and delete respectively *(FORBIDDEN by 42 Norm… Before day06)*
1. malloc and delete use the #include <stdlib.h> > directive

Process for allocation
1. Declare pointer variable to point to allocated heap space
2. Call to allocate the correct number of space in memory
3. Initialize array (If you can)

Example 1 – How to dynamically allocate in C

```
//declare a pointer variable to point to allocated heap space
int     *p_array;
double *d_array;

//call malloc to allocate that appropriate number of bytes for the array

p_array = (int *)malloc(sizeof(int)*50);       // allocate 50 ints
d_array = (int *)malloc(sizeof(double)*100);  // allocate 100 doubles

// always CHECK RETURN VALUE of functions and HANDLE ERROR return values

if(p_array == NULL)
{
  printf("malloc of size %d failed!\n", 50  exit(1);
}
```

Passing array as a parameter
When passing an array as a parameter, this:
```
      void arraytest(int a[])
```
means exactly the same as:
```
      void arraytest(int *a)
```

Dynamically changing the size of an array
(FORBIDDEN by 42 Norm… Before day06)

```
  /* Initial memory allocation, use (char *c) to cast as char array */
  str = (char *) malloc(15);

  /* Reallocating memory */
  str = (char *) realloc(str, 25);
  free(str);
```

Lesson 5 – Pointers

- when you declare a pointer, it's not "a pointer of the type int" but rather "a pointer of the type point to int"
- A pointer is: "An object that holds the memory address of another object" So a pointer is Geff pointing to the box he will put your body in. A Variable is Geff storing your body in the box.
- So it's technically not a variable. Instead of making space in memory (stack), it just waits and points to an area of memory until you decide to use it.
- The reason why we use pointers when creating arrays is because pointers point to an area of memory and allow use to read/manage it accordingly

HOW TO USE POINTERS:
1. Declare a pointer using:
   char * ptr;   //decide variable type
2. Point a pointer to an area of memory
   ptr = my_name;   //my_name is an string containing "Luyanda"
3. Increment a pointer to shift UP in the memory area
   ptr++;   //ptr will now contain "uyanda"
4. Increment a pointer to shift DOWN in the memory area
   ptr--;   //ptr will now contain "Luyanda" again

C strings

- in the c language, strings are a special case of an array
- they are an array of characters terminated by a zero value ('\0')
- char s[ ] = {'s','t','r','i','n','g',0}
  is the same as
  char s[ ] = "string".
  Use the second, it's more common

Pointer to a pointer (Pass by reference)

A pointer to a pointer is a form of multiple indirection, or a chain of pointers

```
Pointer  ⟶  Pointer  ⟶  Variable
```

```
#include <stdio.h>
int main(void)
{
    int **a;
    **a = 42;
    Printf("%i\n",**a);        // prints "42"
}
```

In C, double pointers are more common, since C does not have a reference type. Therefore, you also use a pointer to *"pass by reference."*. This is commonly done in places where we want to adjust the contents of memory or want to return specific information from a function (such as a pointer to a different area in memory).

Remember that in C, a function CANNOT *use & change simultaneously* the value of one of its parameters. Functions destroy all variables used once the compiler reaches the end of a function and exits. In C, we would instead try to set a pointer within a function to point to the area of memory of out parameter and THEN only can we access the memory area to change the parameters value.

Let me show you this concept using a simple function that joins two names:

```c
char    *my_function(char *first_name, char *second_name)
{
    unsigned int size;
    char full_names[40];
    //concatenate first and second names
    strcat(new_name, first_name);
    strcat(new_name, " ");
    strcat(new_name, second_name);

    return(full_names);
}
```

- In the above functions, the areas where first_name and second_name are stored will remain untouched. Even if we change things:
  first_name = Mufaro;
  after the function executes, first_name will return to the original value it was when it entered the function.
- To change first_name in our function without using a new pointer, our function prototype would need to look like this instead:

  ```c
  char    *my_function(char **first_name, char *second_name)
  ```
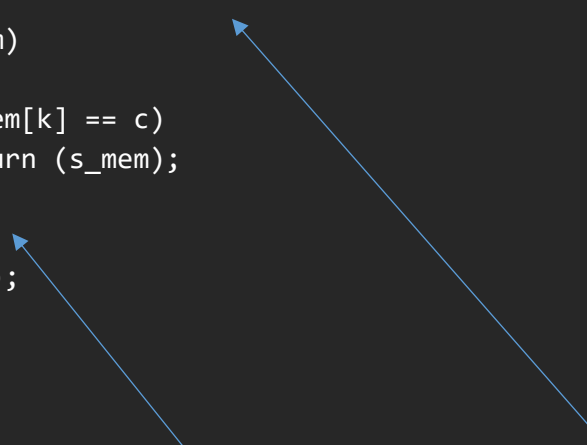 Now we would be able to use:
      first_name = "Arno";

Pointers and memory (memcmp, memcpy, memchr)

Consider an implementation of the memchr function:

```c
void    *memchr(const void *s, int c , size_t n)
{
    /* man page requirements:
        - scan memory for character
        - memory area is interpreted as unsigned char
        HOW TO RETURN A POINTER TO A SPECIFIC LOCATION OF AN ARRAY
            1 assign a pointer to the array in memory
            2 increment the pointer to the position you need
     */
    size_t k;
    unsigned char *s_mem;            //memory area interpreted as unsigned char
    s_mem = (unsigned char*)s;
    k = 0;
    while(*s_mem)
    {
        if (s_mem[k] == c)
            return (s_mem);
        s_mem++;
    }
    return(NULL);
}
```

Pointer arithmetic:
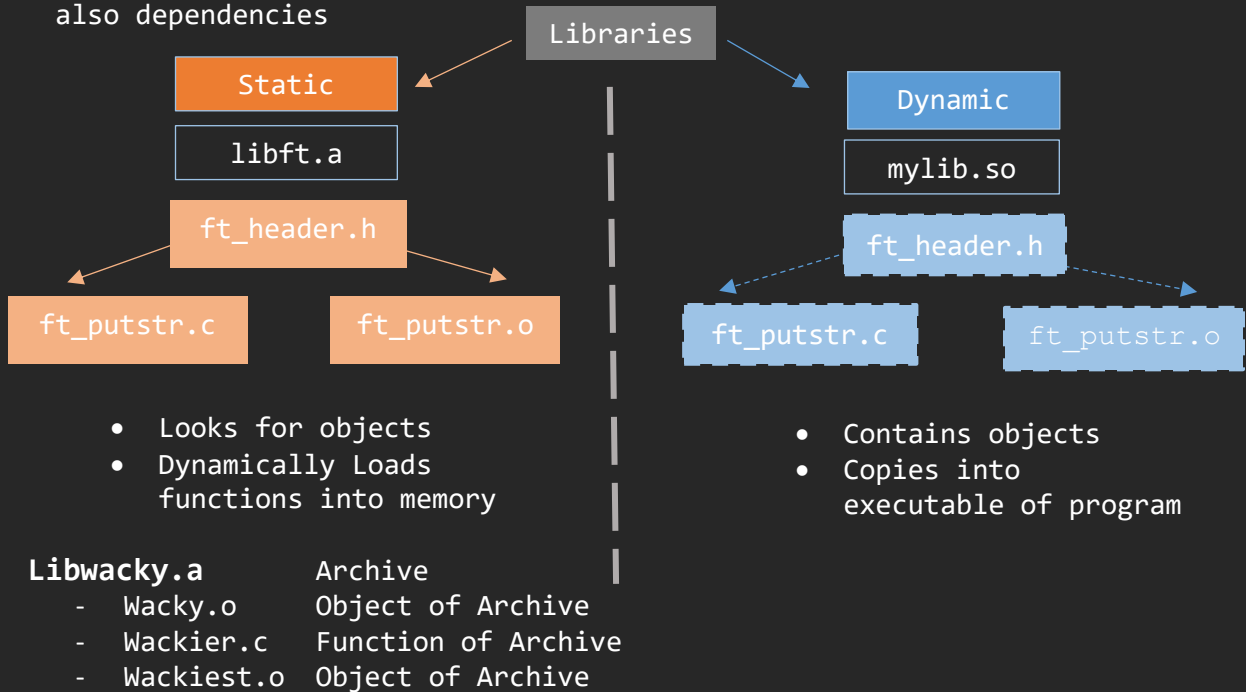*The purpose of the function is to return a pointer to area of memory where char c is found. To get to c, we need to increment our pointer until c is found and then we can return it (refer back to PRO tips!)*

Pointer Casting/ Memory interpretation:
Since the memory area *s is of no specific type (void), we can declare a pointer and point it to this memory area to read or change memory contents

Lesson 6: Libraries, Objects & Header files

- Libraries essentially bundle code/functions/objects together to reuse and access everything easily
- Archives describe how everything is housed
- These are further separated into Static (.a files) & Dynamic/Shared Libraries (.o files)
- Dependencies are created between the Program and the archive. There are also dependencies

```
                              Libraries

        Static                                      Dynamic

        libft.a                                     mylib.so

      ft_header.h                                 ft_header.h

 ft_putstr.c    ft_putstr.o              ft_putstr.c    ft_putstr.o
```

- Looks for objects
- Dynamically Loads functions into memory

- Contains objects
- Copies into executable of program

Libwacky.a   Archive
- Wacky.o   Object of Archive
- Wackier.c   Function of Archive
- Wackiest.o   Object of Archive

How to create Objects

- These are usually created from the .c extension, .c files which are put through the compiler are used to create object files.
- The command line or shell scripts are used to execute the different calls we will need to create object files, creating a library, creating an index in the library and even removing all the object files once they have been created
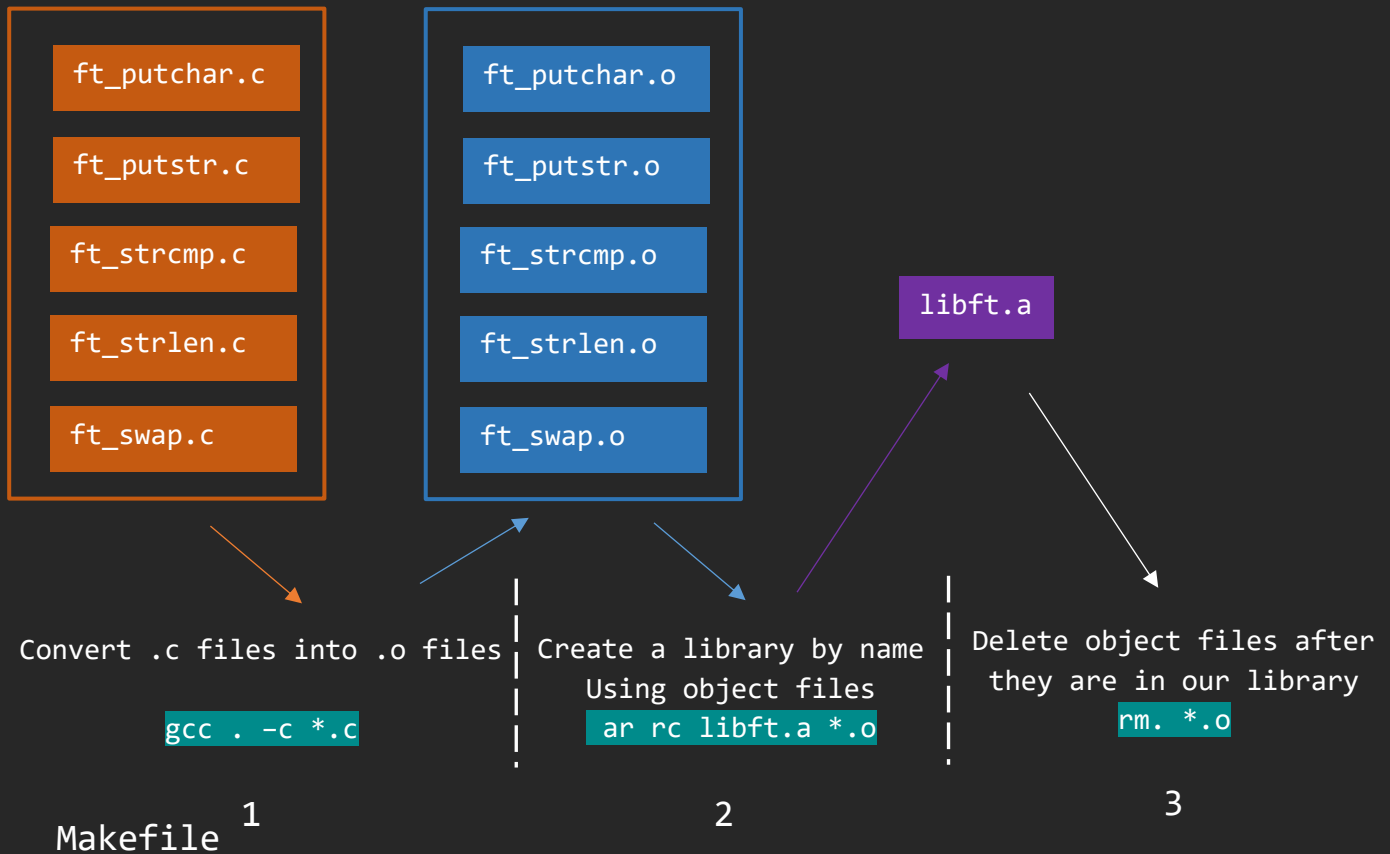
Example – Some shell script to help build up a library (with .c files located in current directory)

`lib_creator.sh`

```
#!/bin/sh
gcc . -c *.c          //Create object files from ALL .c files in current directory
ar rc libft.a *.o     //Create a library by name
ranlib libft.a        //Creates index in library (Not always needed)
rm. * .o              //remove/delete all object files
```

- The ar tool can also help us list the files inside a library
  `ar rc libft.a *.o`

I - Compilation steps (Using a shell file & shell commands)

ft_putchar.c	ft_putchar.o
ft_putstr.c	ft_putstr.o
ft_strcmp.c	ft_strcmp.o
ft_strlen.c	ft_strlen.o
ft_swap.c	ft_swap.o

libft.a

Convert .c files into .o files

`gcc . -c *.c`

Create a library by name Using object files

`ar rc libft.a *.o`

Delete object files after they are in our library

`rm. *.o`

1   2   3

Makefile

- These compilation steps can be put into a file that lets us control which specific step numbers we want to use (Just like commands in the terminal)
- This is called using a makefile. Using the commands is called make commands
- To shorten our compile commands, we can put things like the names or directory paths of files in variables and create rules that basically use the shortened versions of those commands

Example:
    gcc . -c ft_putchar.c ft_putstr.c ft_strcmp.c ft_strlen.c ft_swap.c

could be shortened to…
    gcc -c $(SRC)
where the variable $(SRC) contains all the names of your function .c files

1
2
3

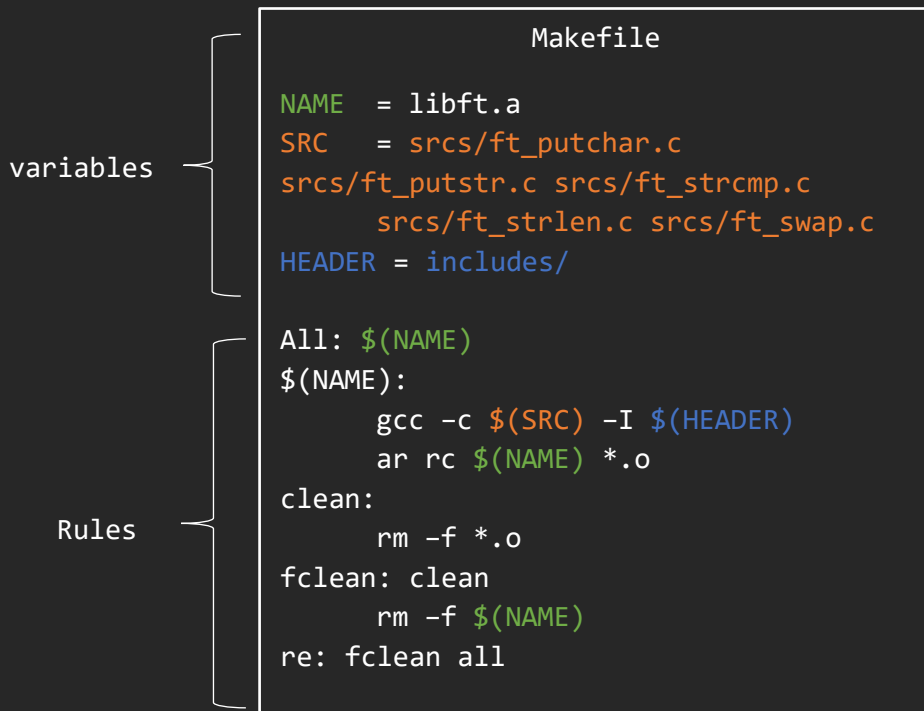- To run each rule individually you need to use make + rule_name
Example:
    rm. *.o
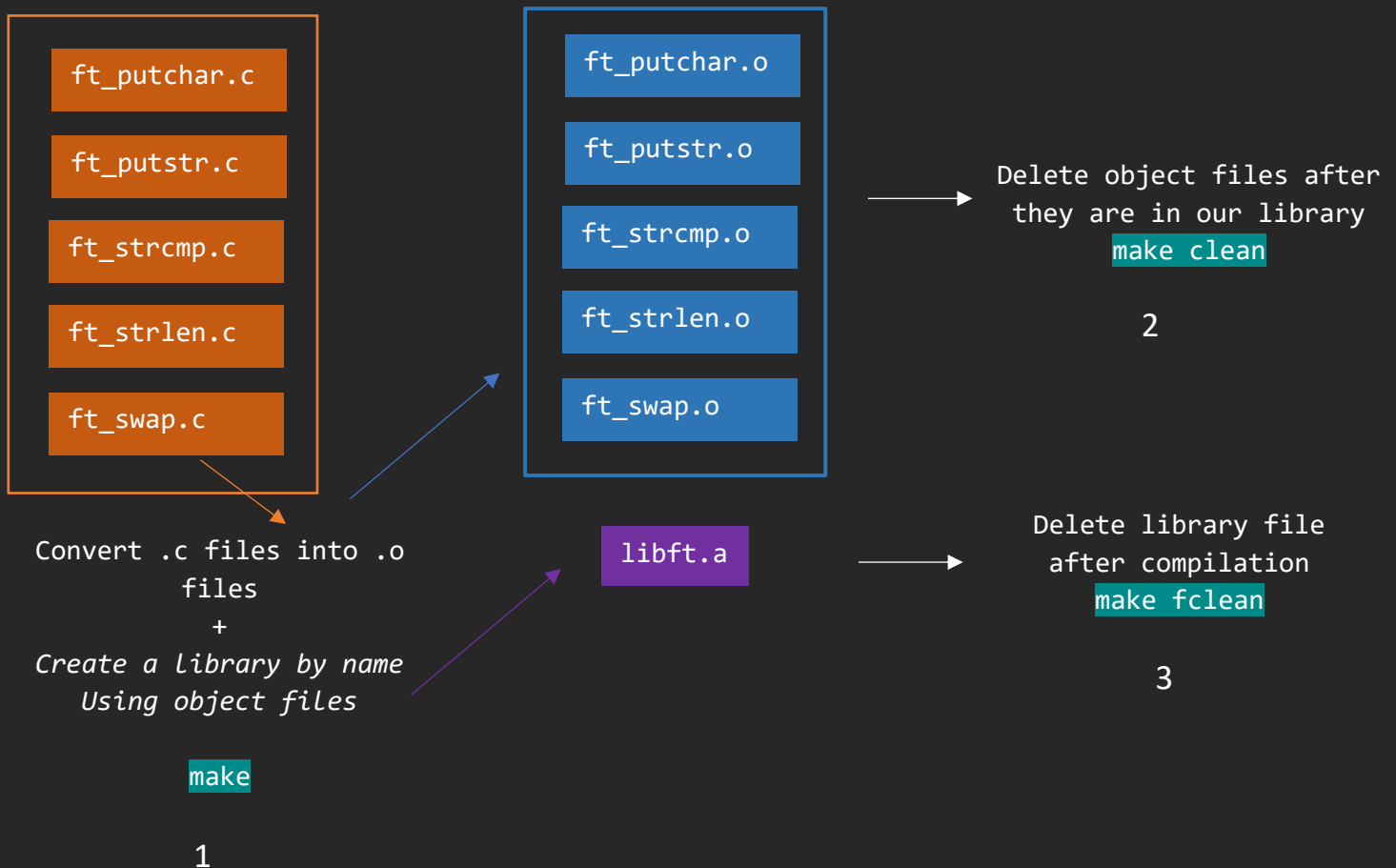would be run using…
    `make clean`

Example:

```
                        Makefile

NAME  = libft.a
SRC   = srcs/ft_putchar.c
srcs/ft_putstr.c srcs/ft_strcmp.c
      srcs/ft_strlen.c srcs/ft_swap.c
HEADER = includes/


All: $(NAME)
$(NAME):
      gcc -c $(SRC) -I $(HEADER)
      ar rc $(NAME) *.o
clean:
      rm -f *.o
fclean: clean
      rm -f $(NAME)
re: fclean all
```

variables

Rules

II - Compilation steps (Using a makefile & makefile commands)

| ft_putchar.c |
| ft_putstr.c |
| ft_strcmp.c |
| ft_strlen.c |
| ft_swap.c |

| ft_putchar.o |
| ft_putstr.o |
| ft_strcmp.o |
| ft_strlen.o |
| ft_swap.o |

Delete object files after
they are in our library
`make clean`

2

Convert .c files into .o
files
+
*Create a library by name
Using object files*

`make`

1

libft.a

Delete library file
after compilation
`make fclean`

3

Header files

- These files contain prototypes for the functions we might want to use but not have to type again and again (i.e. ft_putchar)
- We would define the prototypes for the functions here but usually the logic with the algorithms is contained with a dependant .c file
- So if I create a ft_header.h file with my prototypes, I must create a corresponding ft_functions.c file with my algorithms
- In my ft_functions.c file, I should have an #include "ft_header.h" line to let my compiler know to access prototypes in the ft_header.h header file

<table>
<tr><td>

ft_header.h

```
#ifndef FT_HEADER_H
# define FT_HEADER_H

Void ft_putchar(cahr c);
Void ft_putstr(char *c);
Int  ft_strcmp(char *s1, char *s2);
Int  ft_strlen(char *str);
Void ft_swap(cahr c);

#endif
```

</td><td>

ft_functions.c

```
#include <unistd.h>
Void ft_putchar(cahr c)
{
        Write(1, &c, 1);
}

Void ft_putstr(char *c)
{
//Blah blah blah
}
…
```

</td></tr>
</table>

Using #ifndef … #endif helps us avoid prototype conflicts.

#ifndef
- "If not defined" or if (ft_header == false)
- FT_HEADER is not defined
- So if your compiler cannot find the prototypes anywhere else in your .c files, it will just use the one in ft_header
  (USE THIS ONE, PREVENTS DOUBLE INCLUDES)

#ifdef
- "If defined" or if (ft_header == true)
- FT_HEADER is defined
- So if your compiler can find another instance of one your prototypes, it will just skip over the one in ft_header.h and use the newly defined one elsewhere.

Lesson 7: How to use static Libraries (.a files)

(Useful for using libft in future projects)

- Remember that libraries are separated into Static (.a files) &
  Dynamic/Shared Libraries (.o files)
- To make use of a library we need two things:
     1. A copy of the .a library executable
     2. A header file containing prototypes for needed functions

Example: Driver program that uses a static library

```
                        ft_function.c

#include <stdio.h>

void print_stuff(void)
{
     printf("print_stuff called from static library");
}
```

```
                    myheader.h

 void print_stuff(void);
```

1. Compile using these commands:

```
gcc -c ft_function.c -o ft_function.o
ar rcs mylib.a ft_function.o
```

Compile library files
  Create static library

```
mylib.a
```

Lets test a program that uses our newly created static library

(Remember to include "myheader.h")

```
                        driver.c

 #include "myheader.h"

 void main()
 {
      print_stuff();
 }
```

2. Compile the driver program:

```
gcc -c driver.c -o driver.o
```

3. Link the driver program to the static library:

```
gcc -o driver driver.o function.o -I ./ -L. -lft
```
(-L. instructs the compiler to look in the current directory for the library)

4. Run the driver program:

```
./driver
print_stuff called from static library
```

Lesson 8: Makefiles

- To make better makefiles we need to understand dependencies a bit better
- We also need a better understanding of Linking, Compiling and Including
- Regarding the former, I HIGHLY recommend drawing a quick dependency map using pen and paper. If you are trying to create a library (or code in C++) these will save you hours when you mess up can't understand why your .cpp files don't have access to certain headers
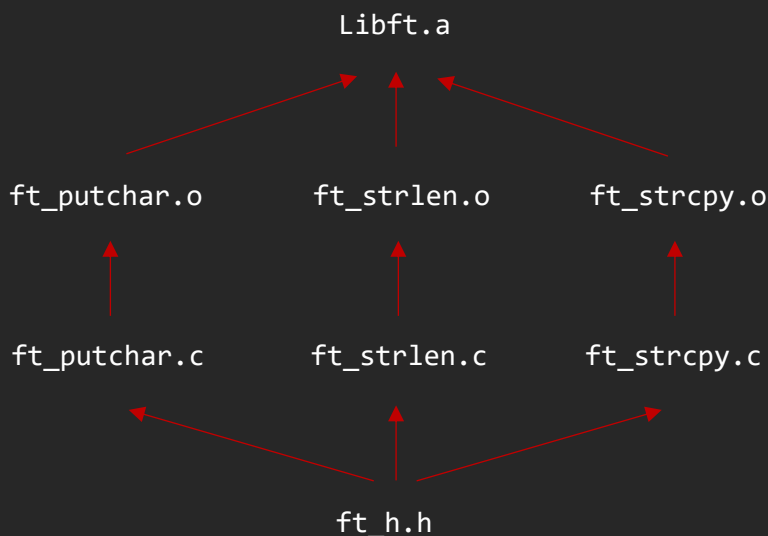
Dependencies

- In C, libraries have dependencies: some files/programs need other files to work correctly
- If you've ever been heroic and attempted to install programs in Linux operating using the compile from source code method, you should understand why this is a big deal.
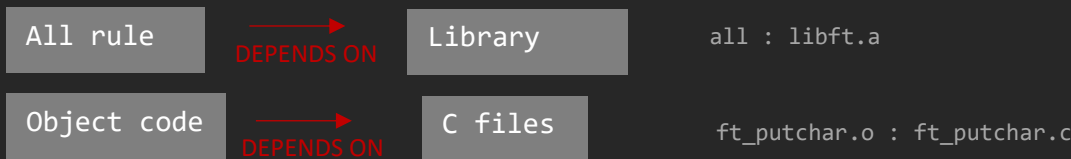
My condolences to you if some dude with a long beard on stack overflow convinced you that this is ever a good idea. It's not. It only ends with you staring at a program that takes up space and does not actually work because of the 15 different dependencies you still need to install after wasting 6 hours of your life trying to install the 100 other dependencies that the program requires. I need new hobbies.

Library	DEPENDS ON →	Object Code	DEPENDS ON →	C files	DEPENDS ON →	Headers
libft.a		ft_putchar.o		ft_putchar.c		ft_h.h

Example: Dependency map for libft.a

```
                     Libft.a
                    ↗   ↑   ↖

ft_putchar.o      ft_strlen.o      ft_strcpy.o

     ↑                ↑                ↑

ft_putchar.c      ft_strlen.c      ft_strcpy.c
        ↖             ↑              ↗
                    ft_h.h
```

- The second part of dependencies is how the work in makefiles
- In makefiles, the colon indicates where a dependency could possibly be

| All rule | →DEPENDS ON→ | Library | all : libft.a |

| Object code | →DEPENDS ON→ | C files | ft_putchar.o : ft_putchar.c |

- You can also create multiple dependencies across a colon. For example, this is how you could add a header file in an existing dependency rule:

| Object code | →DEPENDS ON→ | C files | →DEPENDS ON→ | Headers | ft_putchar.o : ft_putchar.c ft_h.h |

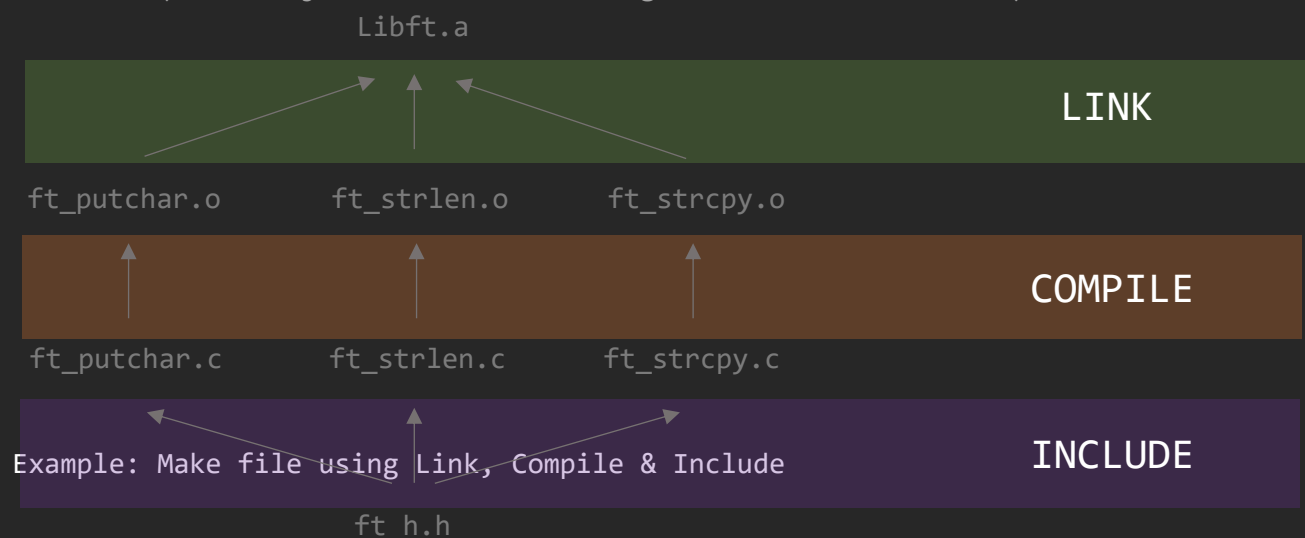Example: Library dependencies in a makefile

```
all: libft.a

ft_putchar.o: ft_putchar.c ft_h.h
```

Linking, Compiling, Including

- Now that we understand dependencies, we can do compilation steps. Properly this time

3 Steps in Compilation

1. INCLUDE
   (Stating the header files we need. These can be self-created, system or packaged)

2. COMPILE
   (Turning the code you have written into machine code. It's so that the computer hardware can read our code)

3. LINK
   (turns object files into a single archive/ executable)

Libft.a

| | LINK |

ft_putchar.o   ft_strlen.o   ft_strcpy.o

| | COMPILE |

ft_putchar.c   ft_strlen.c   ft_strcpy.c

| | INCLUDE |

Example: Make file using Link, Compile & Include

ft_h.h

```
NAME =          objects_SDL
FILENAMES =     Game.cpp Main.c TextureManager.c GameObject.c
_DEPS =         Game.h TextureManager.h GameObject.h
DEPS =          $(patsubst %,$(HDIR)/%,$(_DEPS))

all: $(NAME)

$(ODIR)/%.o: $(SDIR)/%.c $(DEPS)
        $(CC) $< $(HFLAGS) $(IFLAGS) $(CFLAGS) $(LFLAGS) -c -o $@
$(NAME): $(OBJ)
    @echo       Object files generated!
    @echo       Compiling library...
        $(CC) $^ $(HFLAGS) $(IFLAGS) $(CFLAGS) $(LFLAGS) -o $@
    @echo       Done! Type \"make run\" to run executable
```

Explanation of dependencies:

`all: $(NAME)`	*The "all" rule depends on the library/ program objects_SDL*
`$(ODIR)/%.o: $(SDIR)/%.c`	*The "create .o files" rule depends on the .c files with similar names (i.e. ft_putchar.o depends on ft_putchar.c)*
`$(DEPS)`	*The c files in the above rule depend on the header files in the variable called _DEPS*

Lesson 9: Arguments

- This is a super convenient feature to use at compilation for easy testing and to make your programs easier for users to use
- Instead if creating a printf main testing function, you could rather bake in arguments so that you can test at the ./a.out compilation step

argc   - Argument COUNT. This counts the number of entered arguments (i.e. the number of strings entered by user)
           - Is always +1 higher than the actual number, this is because the program name (./a.out) takes up the first index

**argv   - Argument VECTOR. This is a 2D array that holds arguments entered as strings. Is *A NULL POINTER*

-
0	./a.out
1	Hello
2	There!

Example: A program to reverse a string inputted by the user

<table>
<tr><td>

WITHOUT arguments

```
void  ft_putchar(char c);
void  ft_putstr(char *c);
char  *ft_strrev(char *c);


int   main(void)
{
      char *str;
      ft_putstr("Please enter your
      string:");
      scanf("%s",str);
      ft_putstr(ft_strrev(str));
      return (0);
}




Terminal:
$> gcc Reverse_Prog.c
$> ./a.out
$> Please enter your string:
$> ReverseMe
$> eMesreveR
```

</td><td>

WITH arguments

```
void  ft_putchar(char c);
void  ft_putstr(char *c);
char  ft_strrev(char *c);


int   main(int argc, **argv)
{
      char *str;
      int k;

      k = 1;
      while (argc > 1)
      {
            ft_putstr(ft_strrev(argv[k]));
            --argc;
            k++;
      }
      ft_putchar('\n');
      return (0);
}


Terminal:
$> gcc Reverse_Prog.c
$> ./a.out Reverse Me
$> eMesreveR
```

</td></tr>
</table>

HOW TO PRINT ARGUMENTS

- Remember to ALWAYS use both argc & argv in your main function to avoid a compiling error (caused by unused parameters/variables).

WHOLE ARGUMENTS (as _strings_)

```
int k;

k = 1;
if (argc > 1)
while (k < argc)
{
        ft_putstr(argv[k]);
        ft_putchar('\n');
        k++;
}
```

(I prefer this method to avoid spaghetti code)

WHOLE ARGUMENTS (as _char array_)

```
int k;
int m;

k = 1;
m = 0;
if (argc > 1)
while (k < argc)
{
   while (argv[k][m] != '\0')
   {
       ft_putchar(argv[k][m]);
       m++;
   }
   k++;
   ft_putchar('\n');
}
```

(>.<) FML

PRO TIP: Use argc & argv as parameters in functions other than int main

In C, argv works like a char array where the final element is guaranteed to be a null pointer. Pass them as normal if you want to use them outside main.

```
int   main(int argc, **argv)
{
}
```
Is the same as…
```
char  some_or_other_function(int argc, **argv);
```

Is the same as…
```
int   some_or_other_function(int size, char **argspooppoop);
```

Is the same as…
```
void  some_or_other_function(int size_t, char *argblahlah[]);
```

Lesson 10: Pointers to functions

- This is a great way of managing and debugging complex code
- Instead of having to fix a written function that appears 100 times, you can basically alter your function once in a single location and fix all your problems in the original function definition.

Example : fix a function that originally created cows with 3 legs

```c
void  create_cows(int *f)(int **legs, int **head, char **color)
{
    legs = 3;          //We should give a cow 4 legs instead of 3
    head = 1;
    color = "black and white"
}

int main ()
{
    void (*fp)(int **l, int **h, char **c);     //This creates a pointer to
                                                hold our pointer to function

    fp = ft_dostuff;

    int no_legs = 0;
    int no_heads = 0;
    char str_color = 0;
    fp(no_legs, no_heads, str_color);
}
```

- In this example we can see that in the future we might create more cows and use the create_cows function.
- If we were coding a farm simulator, we would probably call the function thousands of times, but to neaten our code we could call it using f(m,n,p); where m, n and p are different values
- Instead of wasting space in memory and storing & executing the function thousands of times, we could just point to it and execute to create a cow when needed.
- Even when we fix our original function, we only need to change 3 to 4 and all the other pointers to our function will adjust themselves accordingly.

TESTING: HOW TO USE A POINTER TO A FUNCTION IN AN ARGUMENT

```c
void ft_putnbr (int nb);
int main (void)
{
    int a[] = {0,1,2,3,4,5}
    ft_foreach(a, 7, &ft_putnbr);
}
```

```c
int main (void)
{
    char **tab;
    tab = (char **)malloc(12);
    tab[0] = "tutu";
    tab[1] = "tata";
    tab[2] = "toto";
}
```

Lesson 11: Understanding Memory Leaks

- One word. Valgrind. Find out how to install and use this amazing piece of software on your Unix based system. It helps detect memory leaks and with the `-ggdb3` tag when compiling your program, you can pinpoint which functions and what line are causing memory leaks.
- `ggdb3` provides extra debugging information at compile time

Example – testing get_next_line.c

```
make -C libft/ fclean && make -C libft/
gcc -Wall -Wextra -Werror -I libft/includes -o get_next_line.o -c get_next_line.c -ggdb3
gcc -Wall -Wextra -Werror -I libft/includes -o main.o -c main.c
gcc -o test_gnl main.o get_next_line.o -I libft/includes -L libft/ -lft
```
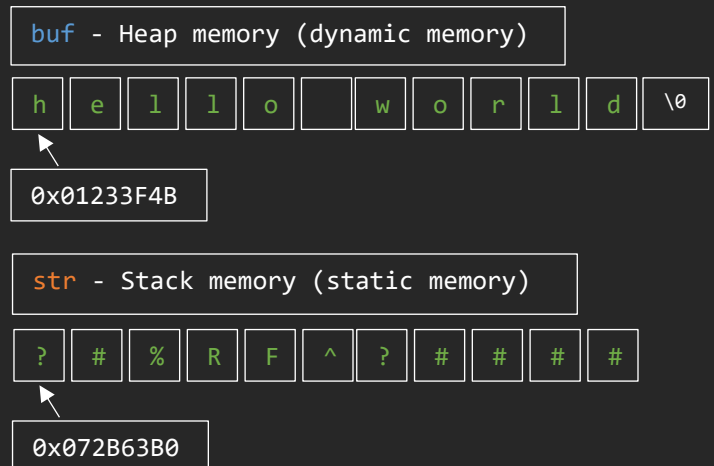
Run the test program (with arguments):
valgrind ./test_gnl bigtest.txt

You should get output that might look like:

```
==24354== LEAK SUMMARY:
==24354==    definitely lost: 74 bytes in 14 blocks
==24354==    indirectly lost: 0 bytes in 0 blocks
==24354==      possibly lost: 0 bytes in 0 blocks
==24354==    still reachable: 1 bytes in 1 blocks
==24354==         suppressed: 0 bytes in 0 blocks
….
==24354== 11 bytes in 1 blocks are definitely lost in loss record 12 of 14
==24354==    at 0x482F228: malloc (vg_replace_malloc.c:299)
==24354==    by 0x108D4B: ft_strnew (ft_strnew.c:21)
==24354==    by 0x108BA9: ft_strjoin (ft_strjoin.c:21)
```

This is what a memory leak might look like:

```
int main (void)
{
    char *buf;
    char *str = "hello world!";
    buf = (char *)malloc(12);
    //insert free(buf); here
    buf = str;
    printf("STRING: %s\n")
    return (0)
}
```

buf - Heap memory (dynamic memory)

| h | e | l | l | o | | w | o | r | l | d | \0 |

0x01233F4B

str - Stack memory (static memory)

| ? | # | % | R | F | ^ | ? | # | # | # | # |

0x072B63B0

We immediately have a leak because of buf. We did not free the memory we allocated to buf before the program exited. str did not cause a leak as it was declared on the stack, and was immediately destroyed once the program exited.

How to avoid memory leaks:

Here is a function that uses this concept to simply use ft_strjoin (function which returns a new string from two smaller ones using malloc) and frees immediately afterwards:

```c
void        ft_swapmem(char **store, char *buf)
{
    char    *tempstore;

    if (*store)
    {
        tempstore = *store;
        *store = ft_strjoin(tempstore, buf); //makes use of malloc
        free(tempstore);
        tempstore = NULL;
    }
}
```

Rule 2: NEVER, EVER, work with the allocated pointer. Use a copy!

This prevents you from accidentally changing the pointer and returning the wrong address over to free. Trust me, this happens A LOT.

```c
    char    *store;
    char    *tempstore;                     //copy/pointer of our memory address
    int     str_len;

    store = (char *)malloc(str_len + 1);
    tempstore = *store;
    while (k > str_len){
        tempstore[k] = ft_toupper(tempstore[k]);
        k++;
    }
    free(store);
}
```

Rule 3: Use more memory than you'll need.

If you need an array 1000 integers long, allocate 2000, and only after you make sure everything else is OK – only then go back and cut it down to 1000.
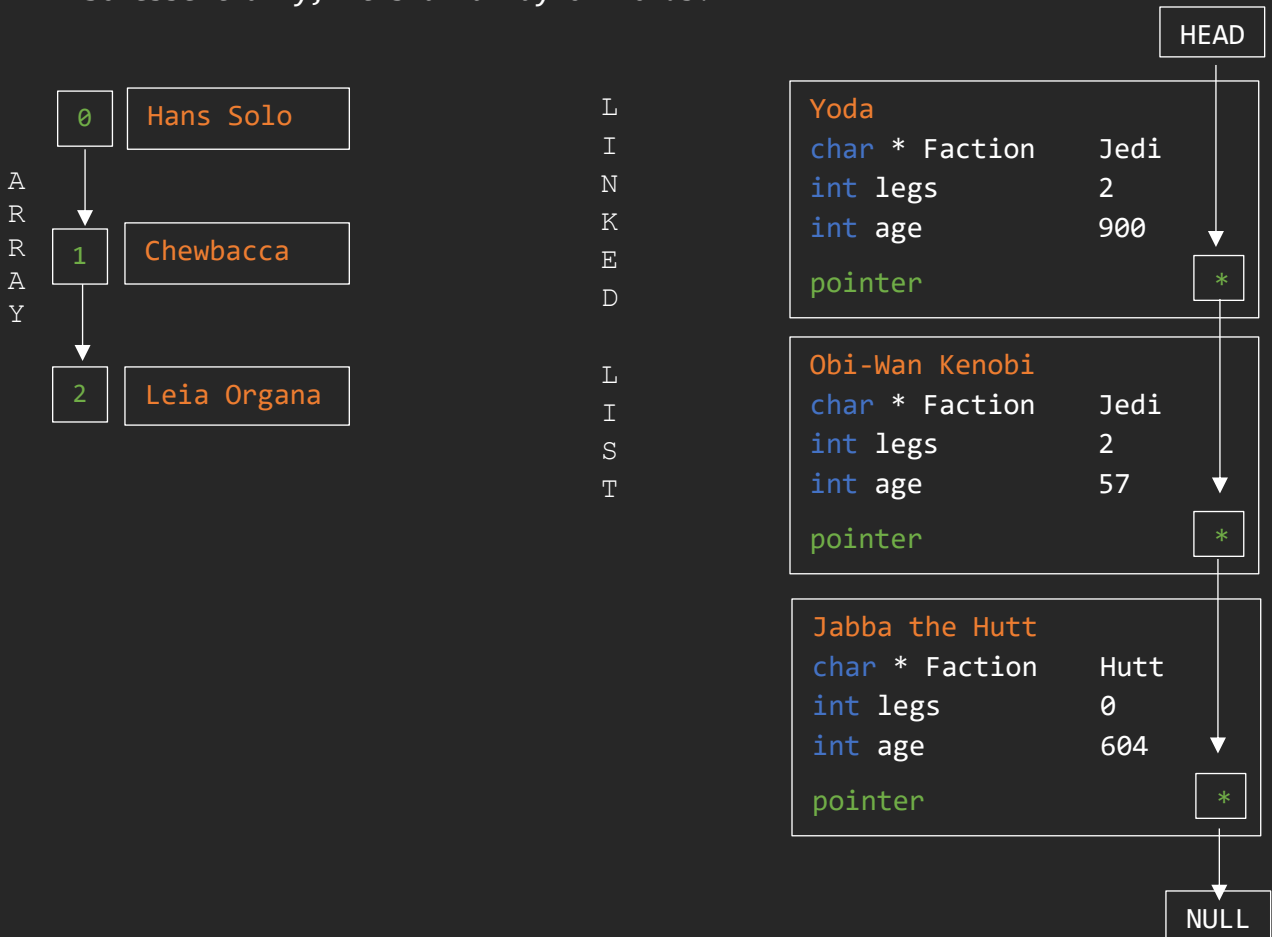
Rule 4: Always carry array length along with you
Wherever and array is, make sure it's length is close by for reference.

Rule 5: Use valgrind!

Lesson 11: Linked Lists

- A linked list is a list constructed using pointers
- It can grow and shrink in size while your program is running
- It can also contain more than a single data type (using nodes)
- So essentially, it's an array on roids.

```
HEAD

A     ┌───┬──────────────┐          L      ┌──────────────────────────────┐
R     │ 0 │ Hans Solo    │          I      │ Yoda                         │
R     └───┴──────────────┘          N      │ char * Faction      Jedi     │
A          │                        K      │ int legs            2        │
Y          ▼                        E      │ int age             900      │
      ┌───┬──────────────┐          D      │                              │
      │ 1 │ Chewbacca    │                 │ pointer              [ * ]    │
      └───┴──────────────┘                 └──────────────────────────────┘
           │                                              │
           ▼                        L      ┌──────────────────────────────┐
      ┌───┬──────────────┐          I      │ Obi-Wan Kenobi               │
      │ 2 │ Leia Organa  │          S      │ char * Faction      Jedi     │
      └───┴──────────────┘          T      │ int legs            2        │
                                           │ int age             57       │
                                           │                              │
                                           │ pointer              [ * ]   │
                                           └──────────────────────────────┘
                                                          │
                                                  ┌──────────────────────────────┐
                                                  │ Jabba the Hutt               │
                                                  │ char * Faction      Hutt     │
                                                  │ int legs            0        │
                                                  │ int age             604      │
                                                  │                              │
                                                  │ pointer              [ * ]   │
                                                  └──────────────────────────────┘
                                                                 │
                                                               NULL
```

- Each box in an array is an index but each box in a linked list is a <u>node</u>. Each node contains a pointer.
- We create nodes using structs in C. A struct is sort of like a class, you can add multiple properties of different data types as show above
- We can add nodes wherever we want to unlike arrays, where we would need to shift other indexes just to insert an element in the middle.
- To fully understand how adding nodes works, we start with what we call the head pointer. It ALWAYS points to the first element in a linked list.

What a node looks like

ft_list.h
```
typedef struct node    s_list
{
     struct s_list    *next;        //Holds address of next node
     Void             *data;        //This could be any random data
}                      t_list
```

In both examples below, begin_list, is a head pointer that will point to the
first node on the list
Add a node at: BEGINNING of the list

ft_list_push_front.h
```
void  ft_list_push_front(t_list **begin_list, void *data)
{
     t_list *cursor;
     cursor = ft_create_elem(data);

     if (*begin_list == NULL)
          *begin_list = cursor;
     else
     {
          cursor->next = *begin_list;
          *begin_list = cursor;
     }
}
```

Create a new node

If head pointer points to NULL -point head pointer to new node

Otherwise, -point new node to existing node -point head pointer to new node.

Add a node at: END of the list

ft_list_push_back.h
```
void  ft_list_push_back(t_list **begin_list, void *data)
{
     t_list *cursor;
     cursor = *begin_list;

     if (*begin_list == NULL)
          *begin_list = ft_create_elem(data);

     while (cursor->next != NULL)
          cursor = cursor->next
     cursor->next = ft_create_elem(data);

}
```

Place a header pointer before first element

If head pointer points to NULL - create new node -point head pointer to new node.

Otherwise, -iterate till end of list -create new node

Few things to remember:
- In our node we have a "void *data" data type. This allows for ANY data type to be inserted in this space.
- The above is convenient as when we test our program we can put in ANY other data type such as an integer or char.

The line:
```
if (*begin_list == NULL)
    *begin_list = cursor;
```
is not always necessary. Sometimes it is helpful to know whether or not our linked list is empty but our create_elem function already accounts for this situation

The line:
```
while (cursor->next != NULL)
    cursor = cursor->next
```
is the standard for looping through or "traversing" our linked list

PRO TIP: What's the difference between `cursor->next` and `cursor.(*next)` ??

Nothing. They're the same. `cursor.(*next)` is C++ syntax sugar.

Lesson ??: Fun stuff

Q: How do I remember the highest integer value for exams? (2,147,483,847)

A: 2 maximum snooker breaks   2   breaks
       A maximum break is 147   147   max
       4 years   48   months
       3 years   36   months
       4 years   48   months

       Or you can declare a variable:

       long int big;
       big = 0x7fffffff;
       'big' now has the value of (2,147,483,847)

Q: WTF is arr<:BUFF_SIZE:> = 0?

A: the same as arr[BUFF_SIZE]= 0. These are called diagraphs. They are
       alternate representations of operators in C languages. Other examples
       include:

DIAGRAPH	EQUIVALENT
<: :>	[ ]
<% %>	{ }
%:	#

Stuff for nerds.

Stuff for nerds I : Space & Time Complexity of an algorithm

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Similarly, *Space* complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input. This usually shows the number of times an algorithm will have to execute in the <u>worst case scenario.</u>
There are some common running times when analyzing an algorithm:

Big O Notation

(In order of speed)

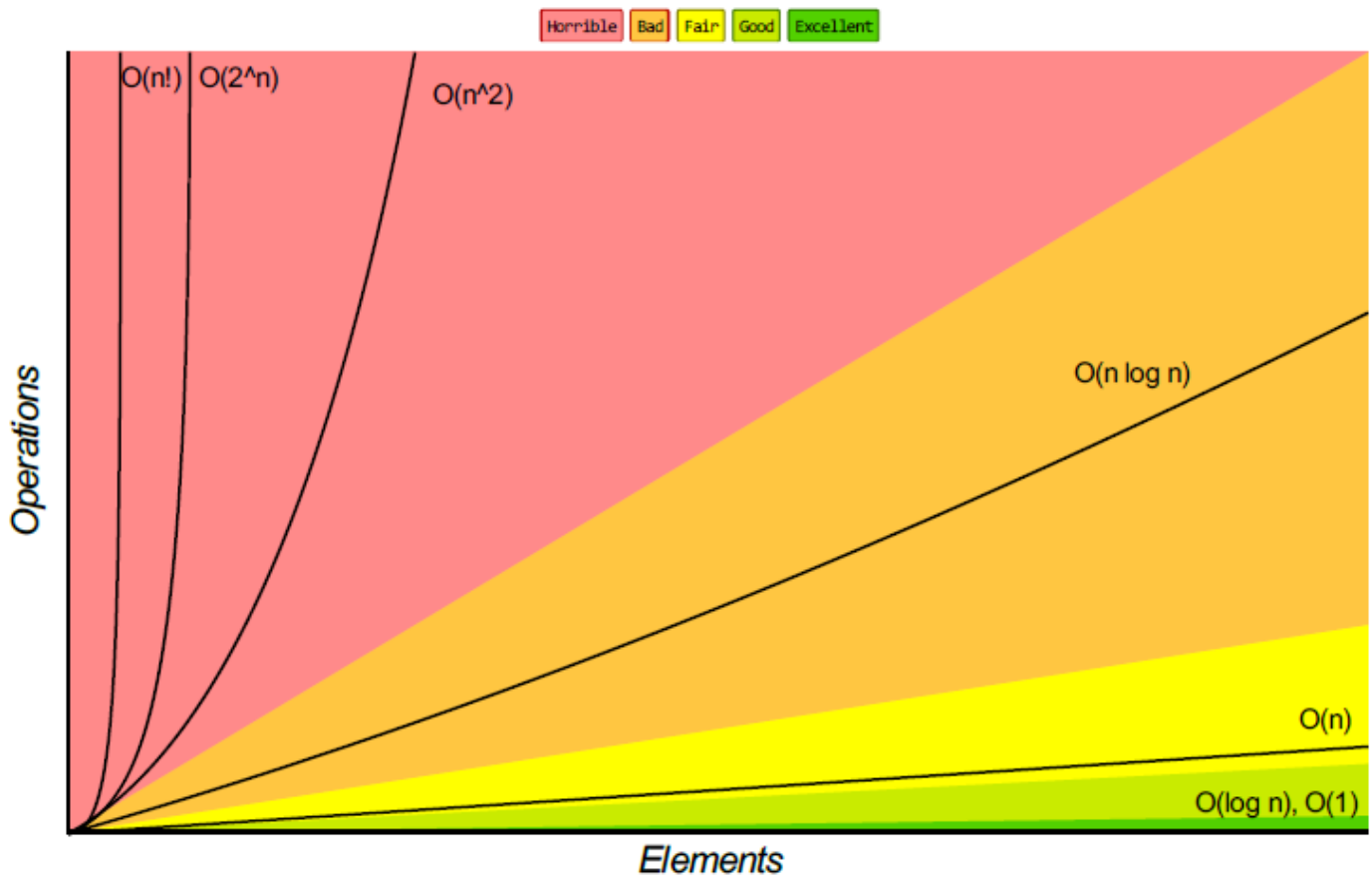|		
$O(1)$ "*Order of 1*"	Constant	Running time is constant, it's not affected by the input size. Example: Adding item to array.
$O(logN)$	Logarithmic	Algorithm that has running time O(log n) is slightly faster than O(n). Commonly, algorithm divides the problem into sub problems with the same size. Example: Binary search algorithm, binary conversion algorithm.
$O(N)$	Linear	When an algorithm accepts n input size, it would perform n operations as well. Example: Linear search
$O(n\ log_n)$	Linearithmic	This running time is often found in "divide & conquer algorithms" which divide the problem into sub problems recursively and then merge them in n time. Example: Merge Sort algorithm.
$O(N^2)$	Quadratic	Look Bubble Sort algorithm! Example: Bubble Sort algorithm (nested loops, swap elements)
$O(N^3)$	Cubic	Same as above
$O(2^n)$	Exponential	Time It is very slow as input get larger, if n = 1000.000, T(n) would be 21000.000. Brute Force algorithm has this running time.
$O(N!)$	Factorial	THE SLOWEST !!! Example : Travel Salesman Problem (TSP)

So if you did ft_fibonacci.c from day04 and tested it, you might have noted that it was REALLY slow, especially at higher values. This is because it has a $O(2^n)$ Time complexity (SUPER inefficient).

So typically, the more data we have, the "N-times" or "N-squared times" etc the algorithm will have to execute in the worst case scenario.

Remember, we always look at the part of the algorithm with the BIGGEST effect on the performance of the algorithm.

Big-O Complexity Chart

Horrible Bad Fair Good Excellent

O(n!) O(2^n)

O(n^2)

O(n log n)

O(n)

O(log n), O(1)

Operations

Elements

:

PRO TIP: TSP for nerds

- A salesman must travel between N cities (in any order) and finishes where he started
- This is an NP-hard problem
- For n cities, you have N-1 Factorial solutions
  (for only 10 cities you have over 180 000 combinations)
- Therefore this has the complexity of $O(n!)$

Stuff for nerds II: Dynamic Programming

(Stuff to know for when we do searches)

- Dynamic programming can make an algorithm more efficient by storing some of the intermediate results (where there are repetitive computations)
- Seems like this is used where we have super slow/inefficient algorithms (like ones that compute in exponential $O(2^n)$ or even factorial $O(n!)$ time)
- Normally 3 approaches to a Dynamic Programming problem:

Recursion

Store (Memoization)

Bottom-up
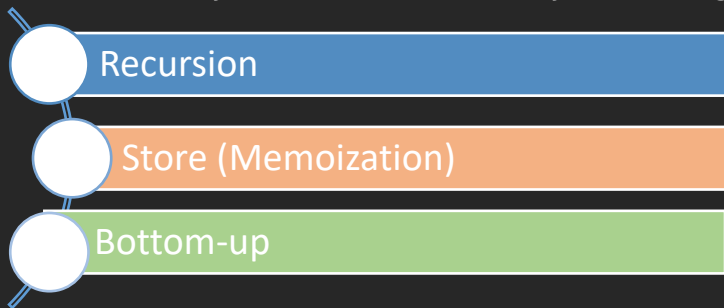
Recursion

- In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.
- But while using recursion, programmers need to be careful to define an exit condition from the function; otherwise it will go into an infinite loop.

The base case = exit condition for function

Steps for writing recursive functions:
1. Define what the recursive function does in English (i.e. the prototype)
2. Think about solving the next smaller sized problem
   Recursion solves a big problem (of size n, say) by solving one or more smaller problems, and using the solutions of the smaller problems, to solve the bigger problem
3. Finally, you should deal with the base case, which is the smallest problem

```
int sum( int arr[], int n )            //1) Function prototype
{
  if ( n == 0 )                        //3) base case
    return 0 ;   // no recursive call
  else
    {
      int small = sum( arr, n - 1 ) ;    //2) solve smaller problem
      return small + arr[ n - 1 ] ;
    }
}
```

Recursion will help solve complex scenarios elegantly and most importantly, will result in readable/understandable code.

Memoization

- Simply storing the results obtained from a recursive solution
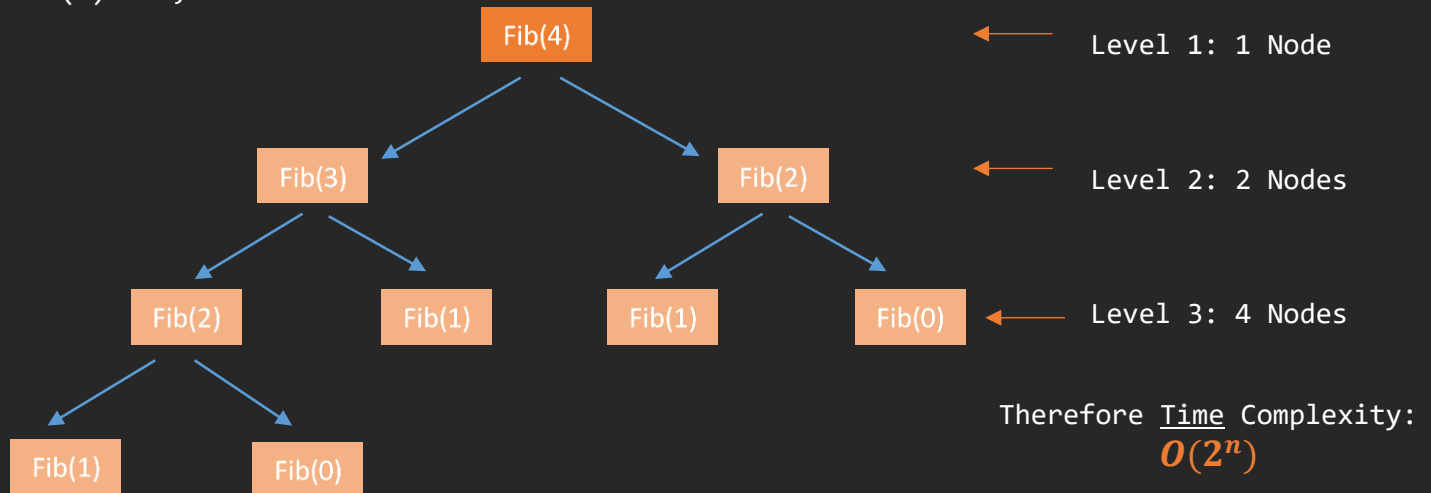- Best example would be the Fibonacci sequence program
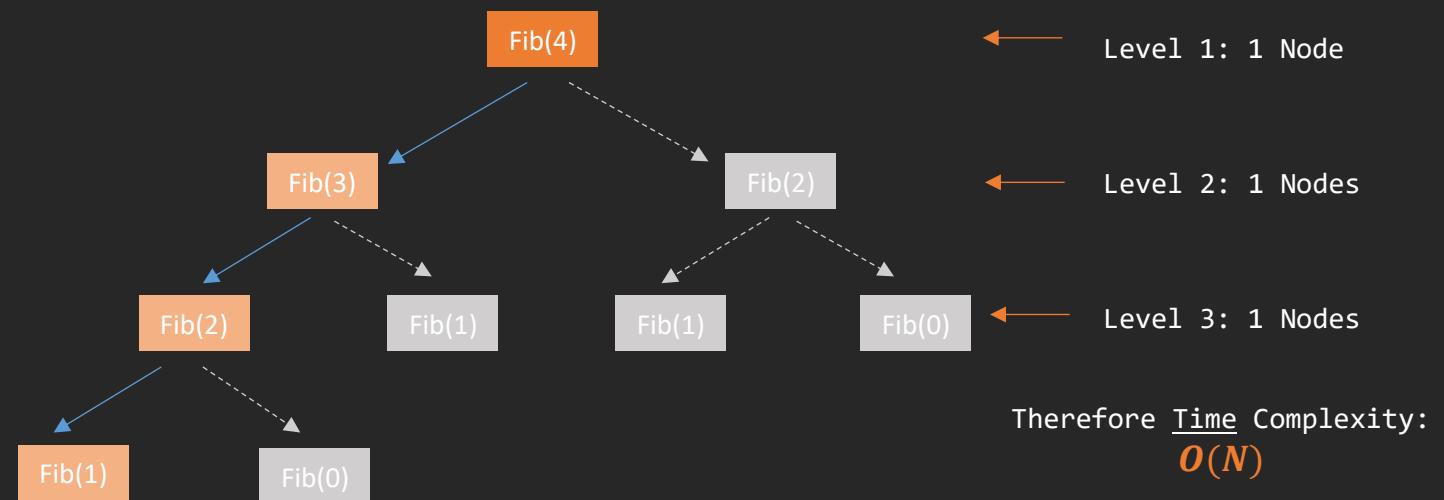
Example:

Fib(n) = Fib(n-1)+ Fib(n-2)

With the starting conditions of:
Fib(0) = 0;
Fib(1) = 1;

Fib(4)   ← Level 1: 1 Node

Fib(3)   Fib(2)   ← Level 2: 2 Nodes

Fib(2)   Fib(1)   Fib(1)   Fib(0)   ← Level 3: 4 Nodes

Fib(1)   Fib(0)

Therefore <u>Time</u> Complexity:
$$O(2^n)$$

A treemap of the fib(4) function (WITHOUT Memoization)

Fib(4)   ← Level 1: 1 Node

Fib(3)   Fib(2)   ← Level 2: 1 Nodes

Fib(2)   Fib(1)   Fib(1)   Fib(0)   ← Level 3: 1 Nodes

Fib(1)   Fib(0)

Therefore <u>Time</u> Complexity:
$$O(N)$$

A treemap of the fib(4) function (WITH Memoization)

Bottom-up

- Better logical approach, instead of recursively trying to find a solution, figure out a relationship on the sequence and store indexes in array

Algorithms for fibonacci using all 3 methods:

Recursion

```
int fib (int n)
{
    int result = NULL;
    if (n == 1)||(n == 2)
        result 1;
    else
        result = fib(n-1) + fib(n-2);
    return result;
}
```

Therefore <u>Time</u> Complexity:
$$O(2^n)$$

Memoization

```
int fib (int n, int *memo)
{
    int result = NULL;
    if (memo[n] != NULL)
    return memo[n]
    if (n == 1)||(n == 2)
        result 1;
    else
        result = fib(n-1) + fib(n-2);
    memo[n]= result;
    return result;
}
```

Therefore <u>Time</u> Complexity:
$$T \leq (2n + 1).O(1)$$

$$= O(N)$$

Bottom-up

```
int fib (int n){
int result = NULL;
    if (n == 1)||(n == 2)
        result 1;
bottom = new int[n+1];          //Define an array
bottom[1]= 1;
bottom[2]= 1;
for (int i=3; i <= n; i++)
        bottom[i]= bottom[i-1]+ bottom[i-2];
    return bottom[n];
}
```

Therefore <u>Time</u> Complexity:
$$= O(N)$$

Stuff for nerds III: Other algorithms

Binary search

- This is an interesting search method that relies on the property that a set is sorted
- This algorithm divides a set in two, compares the search item to the middle of the set, and decides whether it should recursively search the top or bottom of the set.
- Each search level has half the number of results therefore has a logarithmic time complexity $O(log_n)$
- This search can be iterative or recursive (prefer iterative as you can use a single parameter for a function)

Steps:
1. *While left <= right…*
2. *mid = (left + right)/2… OR mid = (left + right)/2 to prevent integer overflow*
3. *Check if x < mid (then adjust right boundary, right = mid-1)*
4. *Else adjust left boundary (left = mid+1)*

```
int binsearch(int nb)
{
      int * arr[nb] , left = 0, right = nb-1, x, mid;
      char * result;
      while (left <= right){
            mid = (left+right)/2;
            if (arr[mid] == x)
                  result = mid ;
            else if (x < arr[mid])                //Check left, adjust right
                  right = mid-1;
            else if (x > arr[mid])                //Check right, adjust left

                  left = mid+1;
      }
      result = '\0'
}
```