Midterm CIS 330 2019

In regards to **bash**...

Be able to go on ix-dev and clone your repo and then make a file, push and commit, then rename that file and re-commit and re-push it.

Example:

```
Mkdir 330Projects
Cd 330Projects
git clone https://alyssakelley97@bitbucket.org/alyssakelley97/uoregon-cis330.git
Touch hello_world.c
```

make the file contents

```
Git add hello_world.c
Git commit -m "Adding the hello_world file" hello_world.c
Git push
Mv hello_world.c goodbye_world.c
Git commit -m "Changing hello_world file name to goodbye_world.c"
Git push
```

(don't Re-add)

Note:

A symbol = name of a function (or variable name)

When you declare -> **week symbol**

- When you do not have a header file included, then you need to include week symbol function declarations at the beginning the file.
- These week symbols are usually all included in your header file.

When you actually create the function -> **strong symbol**

Then how you would compile two files together:

```
gcc -g -o hi.exe hi.o name.o
```

Gcc **-c option flag** - gcc -c compiles source files without linking.

then link with the object files forming the executable.

```
gcc -c [options] [source files]
```

How to compile:

newline.c

```
char *getNewLine()
{
    return("\n");
}
```

```
gcc -g -c newline.c
```

makes the object file newline.o

Linking = takes a bunch of files and combines them together to create an executable.

- You are not running the code yet, the symbols are not associated with any addresses yet.
- Produces the executable, but does not run anything.
- The linker uses the annoying .dSYM directories that they create

Loader = `./exacutable.exe`

- Runs the code

a.out = loader

Creating a library package:

```
ar cr libcool.a name.o newline.o
```

Archive command / cr option / Library name / Object files to package

= You take object files and combine them and then you can use this library. You can use this by

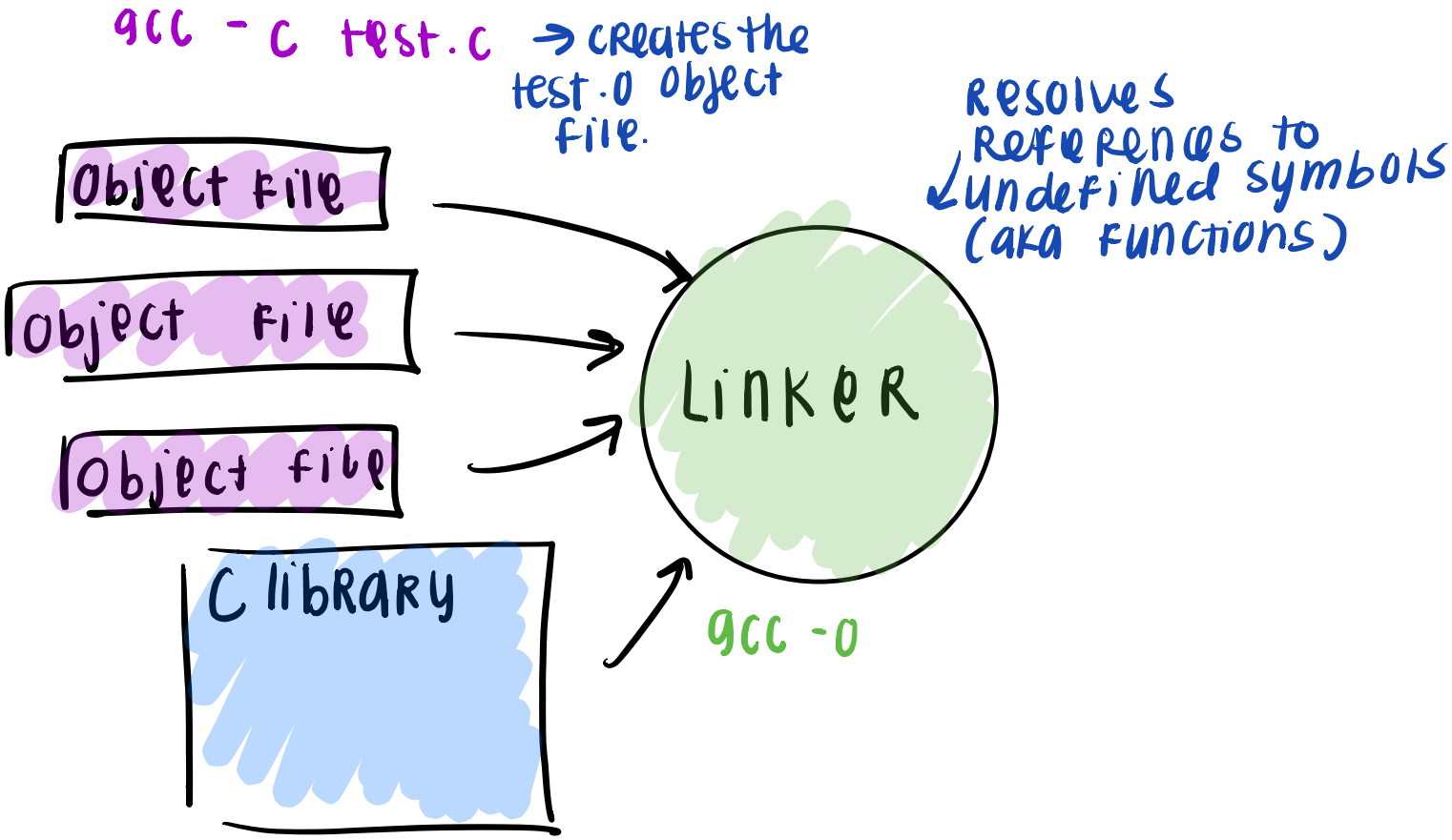
```
gcc -o hi.exe hi.c libcool.a
```

Executable / source file / library package

Dynamic Memory:

C++ = **new** / **delete** are not functions, they are operators.

C = **malloc** / **free** are functions. You get less functionality here. You cannot overwrite these.



Midterm Exam Question:

Static:

Ask you to define something with static memory, no malloc or free. STATIC = NO * NO POINTER.

Static memory is on the stack and it is determined at compile time

3D static array of blocks:

```
Clock arr[5][5][5];
```

Pointer to an array = `Clock *arr[5][5][5];`

```
Clock *clptr = &clock; // if you do not initialize it, this is filled with garbage
```

In C++, if you want to set it to `NULL` use `nulptr;`

Dynamic:

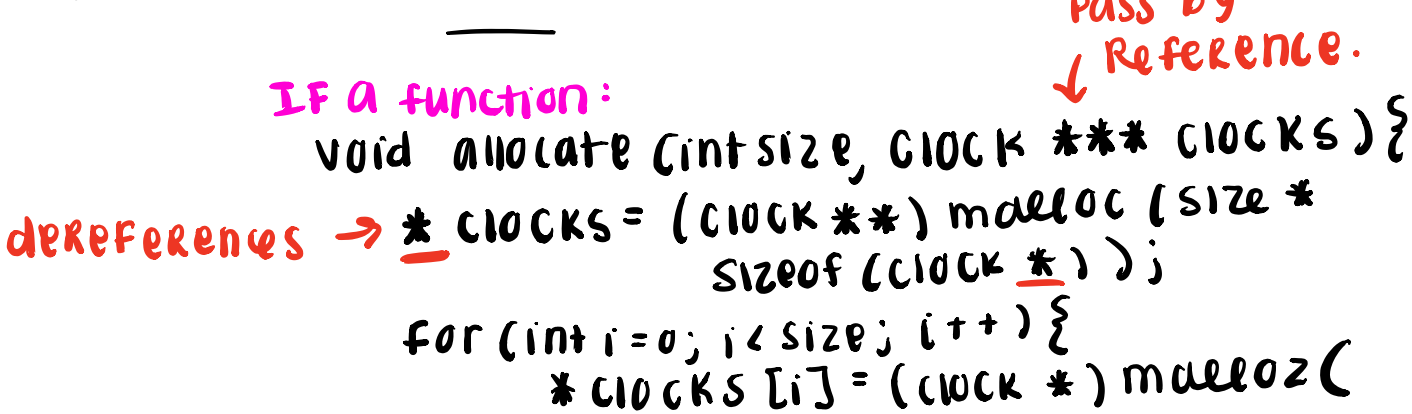
Dynamic memory is created on the heap and is determined at runtime

"Define and initialize a 2D array of Clocks"

```
// 2 x 3 array of clocks, and want to be able to refer to it is clocks[1][2]

typedef char Clock;

Clock **clocks;

clocks = (Clock **) malloc (2 * sizeof(Clock *));

for (int i = 0; i < 3; i++)
{
    clocks[i] = (Clock *) malloc(3 * sizeof(Clock)); // each of these is a pointer to a clock

    memset(clocks[i], 0, 3 * sizeof(Clock)); // initializing it all to 0.

    -- If you don't want to use memset then do... --

    for (int j = 0; j < 3; j++)
        {
            clocks[i][j] = 0;
        }

    free(clocks[i]);
}

free(clocks); // need to free for anything you malloc, so free in the for loop and
// also free in a stand alone line.
```

With string copy... you can have a string put into the data of the array instead of just 0'ing it out...

```
    for (int j = 0; j < 3; j++)
        {
            strcpy(clocks[I][j].stuff, "NO DATA");
        }
```



size * 2 * sizeof(clock));

} }

Understanding the C Compilation process...

The source code goes to the preprocessor which goes to the compiler and that converts it to assembly code and the assembler converts it to object code and then the linker links the object code to the library code and creates the executable files.

Preprocessor - 1) removes the comments and 2) includes the header code in source code and 3) replaces all the macro name with code

```
// Example

#include <stdio.h>
Void main()
{
Printf("hello");
}
```

removes this.

includes the standard library code to bring in functions like printf()

Most common preprocessor macros:

directives

important to remember these are header files and not libraries when refering to them technically.

Stdlib.h - includes the standard library header file
stdio.h - includes the standard io c header file
String.h - includes the standard c string header file

#define = substitutes a preprocessor macro
#include = inserts a particular header from another file

Linker - taking the multiple object code files, and the libraries functions you are using and creates an executable file (.exe) so this is the step on going from the source files to the executable files. The point of this is to have all of the resources needed together so the files can interact. Even if you only have 1 source code file, then you need to know the entry point (aka holding main).

Steps:
1) Compile
2) Link
3) Execute

Pass by reference = passing in the reference (usually seen as &) to the function and it will update that variable value

Pass by value = passing in a copy of the variable so it does not change that variable

for Reference the function declaration and definition has an extra * and in main it is deallocated by using &.

```
void allocate (int size; char **name) {
    *name = (char *) malloc (size * sizeof (char));
}

int main (){
    char *name;  // array for name.
    int name_length = 7; // length (6) + 1
                         (null char)

    allocate (name_length, &name);

    printf ("%s is my name \n", name);
}
```

reference

pointer

memory location for name