GDB Stuff

- Compiling:
  - *-g* token enables the use of GDB
    - *E.g* gcc -g main.c -o main -Wall
- Starting GDB:
  - Use "gdb main" (replace main with the exe name)
    - **Note:** This doesn't run the program, it just starts it in GDB mode
- Setting break points:
  - (gdb) b *line-to-break*
    - (gdb) b 40 When program is running, will break at line 40
  - Deleting break points:
    - Delete *break_point*
- Stepping:
  - (gdb) s
    - Will go ahead and execute the current source line, and then stop execution again before the next source line
- Next:
  - (gdb) n
  - Will continue until the next source line in the current function.
    - This is similar to step, except that if the line about to be executed is a function call, then that function call will be completely executed before execution stops again, whereas with step execution will stop at the first line of the function that is called
- Print:
  - (gdb) p *variable_name*
  - To get the address of a variable:
    - (gdb) p &a
      - This will print the address of variable *a*
- Running program in GDB:
  - (gdb) run
    - If you have command line args, include them after "run"
      - (gdb) "run a b" where a and b are command line args

System Calls to Know:

- Fork()
  - Used to create a new process
  - Returns an int
    - 0 if a child process was created
    - A negative number if creation of child process was unsuccessful
    - A positive value if returned to parent or caller
      - Value contains the process ID of a newly created child process
  - Creates a "copy" of the parent program

        - Has it's own stack and PC tho
- Wait()
    - A way for the parent process to delay its execution until the child is done executing
    - System call won't return until the child has run and exited
    - suspends execution of the calling process until one of its children terminates
- Waitpid()
    - uspends execution of the calling process until a child specified by *pid* argument has changed state.
    - With no arguments, behaves the same way as wait()
- Execvp()
    - Useful when you want to run a program that is different from the calling program
    - given the name of an executable (e.g., wc), and some arguments (e.g., p3.c), it loads code (and static data) from that executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized.
        - Then the OS simply runs that program, passing in any arguments as the argv of that process.
        - Thus, it does not create a new process; rather, it transforms the currently running program (formerly p3) into a different running program (wc)
    - The exex system call replaces the current process with a new program.
    - Using this command, the created child process does not have to run the same program as the parent process does
    - **The exec() family of system calls allows a child to break free from its similarity to its parent and execute an entirely new program.**
    - int execvp (const char *file, char *const argv[]);
        - **file:** points to the file name associated with the file being executed.
          argv: is a null terminated array of character pointers.
- Which()
    - Returns the pathnames of the files (or links) which would be executed in the current environment, had its arguments been given as commands in a strictly POSIX conformant shell. It does this by searching the PATH for executable files matching the names of the arguments
- Where()
    - IDK MAN

String Stuff

- Strcpy()
    - The C library function **char *strcpy(char *dest, const char *src)** copies the string pointed to, by **src** to **dest**.
    - char *strcpy(char *dest, const char *src)
        - **dest** − This is the pointer to the destination array where the content is to be copied.

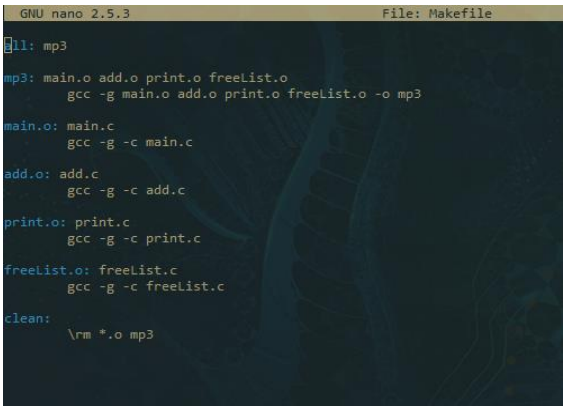
    - **src** – This is the string to be copied.
- Strcmp()
    - int strcmp (const char * str1, const char * str2);
- Strlen()
    - The C library function **size_t strlen(const char *str)** computes the length of the string **str** up to, but not including the terminating null character.
    - size_t strlen(const char *str)
- Making strings in c
    - Strings don't exist in c, instead we have an array of chars
    - Char foo[50]
    - Strcpy(foo, "foo")
    - Len = strlen(foo)
    - Foo[len-1] = '\0'
        - Strings end with \0
        - Null terminator
    - The char *foo* is now a "string"

```
void insert(char *name, int num, char *title, int run_time, int date, char *artist)
{
  node_t *temp, *mp3;

  mp3 = (node_t *) malloc(sizeof(node_t));        // malloc space for MP3
  mp3->name = (char *) malloc(strlen(name) + 1);  // malloc space for name
  mp3->title = (char *)malloc(strlen(title)+ 1);  // malloc space for title
  mp3->artist = (char *)malloc(strlen(artist)+1); //malloc space for artist
  strcpy(mp3->artist, artist);                    // "assign" artist via copy
  strcpy(mp3->name, name);                        // "assign" name via copy
  strcpy(mp3->title, title);                      // "assign" title via copy
  mp3-> date = date;
  mp3->run_time = run_time;
  mp3->data = num;                                // assign data value
  mp3->next = NULL;
  if (head == NULL)
  {
    head = mp3;
    tail = mp3;               // add the first MP3
  }
  else
  {
    temp = head;
    while (temp->next != NULL)
      temp = temp->next;
    temp->next = mp3;
    tail = mp3;         // append to the tail/end
    mp3->prev = temp;
  }
}
```

Struct:

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

typedef struct node
{
  char *name;
  char *artist;
  char *title;
  int date;
  int data;
  int run_time;
  struct node *prev;
  struct node *next;
} node_t;
```

Make File:



Valgrind:

- “valgrind ./program”