Welcome to CSCI 4061

Today

- C Programming Review
  - Memory Allocation
  - Structures
  - Linked List
- Compilation & Make
- Debugger: Basic GDB Commands

Dynamic Memory Allocation

- Sometimes we don't know how much memory we need to allocate beforehand, so we must allocate it on the fly. To do this, we use the `malloc` function.
- `malloc` returns a void-pointer which you must cast to the type of pointer you need. That pointer points to the newly allocated space in memory for your array.

```
int *a = (int *)malloc(sizeof(int)*8);
```

Dynamic Memory Allocation

- It’s important to remember how much memory you allocated so you don’t run off the end of the array.
- Running off the end of a dynamically-allocated array could corrupt data in other parts of your program --> extremely hard to debug!
- You must always keep a pointer that references your newly allocated array so that you can dispose of it when you are done.

Dynamic Memory Allocation

- To free the allocated memory, use the `free()` system call. It takes one argument: a pointer to the allocated memory.
  - `free(a);`
- It's possible to run out of memory. **It is a really really good idea to check `malloc`'s return value every time after allocation to see if its `NULL`.**

```
if(a==NULL){printf("out of memory.");}
```

Structures -Accessing objects with pointers

```
struct point{
int x,y;
};

int main(){
   struct point* p=(struct point*)
   malloc(sizeof(struct point))
   p->x=10;
   p->y=20;
   printf ("x = %d, y = %d\n", p->x, p->y);
}
```

Linked List

```
typedef struct node_t {
int id;
struct node_t * next;
} node;
node *name = (node *) malloc(sizeof(node));
if(name) {
   name->id = 8;// use -> node is a pointer
  name->next = NULL;}
```

How to insert and delete nodes? (pointer operations)

Compilation & Make Tool

- Purpose of Make tool: help a developer with compilation.
- When working on bigger projects it can take a lot of time to recompile all files…
- In most cases only a few files are actually changed by the developer. The make tool keeps track of which files have been changed and recompiles only those files.
- The developer does not have to enter long compiler commands each time--> makes compiling easier!
- The make tool accepts also other types of instructions that can help in automating tasks related to building of programs.

A simple Makefile

- A simple make file might look as follows(last week example):

```
# This is how a comment looks like in a makefile
all:
<--TAB-->gcc helloWorld.c -o helloWorld
clean:
<--TAB-->rm helloWorld
```

- You will find a makefile like this in the test files!

Creating a simple Makefile

- `all` and `clean` are called targets
- Go into the directory where the makefile is located and enter “make” --> the commands listed under all are executed.
- Enter “make clean” --> the commands listed under the clean target are executed
- Try it!

Make - Variables

- We can use variables to remove redundancy in our rules. Take a look at this example:

```
CC = gcc
CFLAGS = -g -Wall
LDFLAGS = -lm (Note: this links the math library)
OBJS = main.o apple.o
myprog: ${OBJS}
${CC} ${LDFLAGS} ${OBJS} -o myprog
main.o: main.c apple.h
${CC} ${CFLAGS} -c main.c
apple.o: apple.c apple.h
${CC} ${CFLAGS} -c apple.c
```

Make - Shortcuts

- If we follow naming conventions, we can do the following:

```
CC = gcc
CFLAGS = -g -Wall
LDFLAGS = -lm
main: main.o apple.o
main.o: main.c apple.h
apple.o: apple.c apple.h
```

- The Make tool uses defaults to automatically compile your program using CC, CFLAGS, and LDFLAGS variables. (Naming conventions must be used for this to work properly - .o, .c, targets.)

Debugger: Basic GDB Commands

- GDB:
  - GNU debugger
  - Command based
- General Commands:
  - run [<args>]: runs selected program with arguments <args>
  - quit: quits gdb
  - s[tep]: step one line, entering called functions
  - b[reak] [<where>]: sets breakpoints. <where> can be a number of this, including a hex address, function name, line number, or relative line offset.
  - d[elete] [<nums>]: deletes breakpoints by number
  - p[rint] [<expr>]: prints out the evaluation of <expr>

GDB-Debug source code program

- Step 1: Compile the source code program(leak.c) with command -g
- Step 2: gdb leak
- Step 3: Set break points
- Step 4: Run arg1, arg2….
- Step 5: Continue(c) or next step(n)

For more information, check GDB cheat sheet:

https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

Debugging tool: Valgrind

- Valgrind
  - Detect memory errors
  - Accesses outside of memory bounds
  - Memory leaks
  - Great for finding errors
  - Try on leak.c in the test files

Eg: valgrind --leak-check=full --show-leak-kinds=all \ --track-origins=yes --verbose \ ./Prog

Static Analysis tool: Splint

- Splint detects
  - Dereferencing a possibly null pointer.
  - Unused variables.
  - Type mismatches, with greater precision and flexibility than provided by C compilers.
  - Memory management errors including uses of dangling references and memory leaks.
  - Eg: Splint leak.c

Exercise

1. Extract exercised_code.tar.gz

2. Implement list.c based on description in the files

- The list.c file contains a linked list which looks like this: 2 -> 6 -> 10
- Take one integer as command line argument and insert it into the list so that the list remains sorted
- Add your code in the insert_list() function

3. Run grade.sh

4. Use Valgrind and Splint to check your code

5. Check rubric.txt for the grading

6. Submit your tar file on Canvas

Questions?