Welcome to CSCI 4061
Today

• C Programming Review
  – Arrays
  – Pointers
  – Memory Allocation
  – Structures
• Compilation & Make
• Debugger: Basic GDB Commands
C Programming Review

• C programming will be used heavily throughout this course.
• If you haven’t programmed in C before, **Right Now is the time to learn!!** There are tutorials on the course website.
• We will review some C basics to refresh our minds.
C Programming Review

Here’s a simple C program:
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf(“Hello World!\n”);
    printf(“argc : %d, argv[0] : %s”, argc, argv[0]);
    return 0;
}

The value argc is the number of the arguments including the command, and argv is an array of strings containing each argument.
Arrays in C

- Arrays are used to group data consecutively in memory and to provide an easy access to them.

```c
int array[10];
int my_ints[] = {4,5,10,27};
int i;
for (i=0;i<4;i++)
    printf("my_ints[%d] is %d", i, my_ints[i]);
```
Pointers in C

• Pointers are used to reference variables by their address instead of by name.
• The *operator is used to dereference a pointer, and the &(address-of) operator is used to give the address of a variable

```c
int m=7;
int *p; //declare a pointer variable
p=&m; //p now points to m
*p=*p+3; //m is now 10
```
Pointers in C

• Don’t get confused about where the * is used! What’s happening in this example?
  - int *p;
  - *p = 1;
Pointers in C

• Don’t get confused about where the * is used! What’s happening in this example?
  – `int *p;` //declaring a pointer variable
  – `*p = 1;` //the value which p points to is now
Arrays & Pointers in C

• All arrays in C can be treated as pointer. This allows us to do pointer arithmetic

```c
int a[6] = {1, 2, 3, 4, 5, 6};
int i;
for(i=0; i<6; i++)
    *(a+i) = *(a+i)+1;
```
Arrays & Pointers in C

int a[8], x;
int *pa;

pa = &a[0]; //pa points to address of a[0], pa = a
x = *pa; //x = contents of pa(a[0] in this case)

a[i]  <-> *(a + i)
&a[i] <-> a+i
pa[i]  <-> *(pa + i)
pa+i   <-> &a[i]
Arrays & Pointers in C

```c
int a[8] = {1,2,3,4,5,6,7,8};
char c[8] = {'a','b','c','d','e','f','g','h'};
int *pa;
char *pc;
pa = &a[0];
pc = &c[0];

for(i=0;i<8;i++)
    printf("%d. pa_add: %p val : %d, pc_add : %p val : %c\n", i, pa+i, *pa+i, pc+i, *pc+i);
```
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.**

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

• Structures allow the bindings of several datatypes.

```c
struct complex_num {
  float real;
  float imaginary;
};
struct complex_num name; // Don’t forget struct!
name.real = 8.888;
name.imaginary = 6.666;
struct complex_num name[8]; // a structure array with 8 elements
name[6].real = 8.888;
name[6].imaginary = 6.666;
```
typedef struct node_t {
    int id;
    struct node_t * next;
} node;

node *name;
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.
Creating a simple Makefile

- Create a file with the name of “makefile” in the directory where the source files are located.
- A simple make file might look as follows:
  ```make
  # This is how a comment looks like in a makefile
  all:
      gcc helloWorld.c -o helloWorld
  clean:
      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!
- NOTE: This is an extremely simple makefile.
- Download test_files2 from the class webpage and extract 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>`
Debugging tool: Valgrind

• Valgrind
  – Detect memory errors
  – Accesses outside of memory bounds
  – Memory leaks
  – Great for finding errors
  – Would only show during harsh test cases
Static Analysis tool: Splint

- Splint detects
  - Dereferencing a possibly null pointer.
  - Using possibly undefined storage or returning storage that is not properly defined.
  - Type mismatches, with greater precision and flexibility than provided by C compilers.
  - Memory management errors including uses of dangling references and memory leaks.
  - Dangerous aliasing.
  - …
Questions?