Lecture 22 (3-16-26)

Libraries, Pointers (C)

Motivating Questions

- What exactly is:
  - a pointer?
  - an array?
  - a string?

Compiling (Review from last lecture)

- Stages:
  1. Preprocessing:
     - expands all pound defines and other things
  2. Compile
     - optimization
     - generates assembly
  3. Assemble
     - assembly to binary code (ELF)
  4. Linking
     - links libraries and other things that are needed
- Makefiles:
  - combine the first three stage and call it "compile"
  - leaves the last step as "linking"

Example

```
/* main.c */

extern void greet(const char*);
// "extern" -> "this func comes from outside this file"

int main(int argc, char *argv[]){
    //TODO
```

```
    for (int i = 1; i < argc; i++){
        greet(argv[i]);
    }
    return 0;
}
```

```
/* library.c */

#include <stdio.h>

void greet(const char *name){
    printf("hello, %s\n", name);
}
```

```
CC=         gcc
CFLAGS=     -Wall -std=gnu9 -g -fPIC
LD=         gcc
LDFLAGS=    -L.

all:        program

greet.dynamic:  main.o libgreet.so  #LINKING
    $(LD) $(LDFLAGS) -o greet.dynamic main.o -lgreet

greet.static:   main.o libgreet.a   #LINKING
    $(LD) $(LDFLAGS) -static -o greet.static main.o -lgreet

program:    main.o library.o.  #LINKING
    $(LD) $(LDFLAGS) -o program main.o library.o

libgreet.so: library.o         #LINKING
    $(LD) $(LDFLAGS) -shared -o libgreet.so library.o

libgreet.a: library.o          #LINKING
    $(AR)  rcs libgreet.a library.o

library.o: library.c              #COMPILING
    $(CC) $(CFLAGS) -c -o library.o library.c

main.o:                           #COMPILING
    $(CC) $(CFLAGS) -c -o main.o main.c
clean:
    rm -f program library.o main.o
```

```
ldd ./program

# this command will show you what libraries your program needs

file program.c

# this command tells you what type of thing program.c is
```

```
env LD_LIBRARY_PATH=. ./greet dynamic sam kiriss matt
```

Pointers

`addresses.c`:

```
/* addresses.c */

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

int main(int argc, char *argv[]) {
    int    i = 321;
    char   c = 'p';
    float  f = 3.14;
    double d = 3.14;

    printf("Value of i = %8d, Address of i = 0x%lx, Size of i = %lu\n", i,
(unsigned long)&i, sizeof(i));
    printf("Value of c = %8c, Address of p = 0x%lx, Size of c = %lu\n", c,
(unsigned long)&c, sizeof(c));
    printf("Value of f = %8f, Address of f = 0x%lx, Size of f = %lu\n", f,
(unsigned long)&f, sizeof(f));
    printf("Value of d = %8lf, Address of d = 0x%lx, Size of d = %lu\n", d,
(unsigned long)&d, sizeof(d));

    return EXIT_SUCCESS;
}
```

- the code above prints very high addresses
- Every variable declaration is **memory allocation**.
- A ***pointer*** is an ***integer*** whose ***value*** is an ***address***

- When we **dereference** a pointer, we access the **value that corresponds to the address** (*p)

pointers.c

```
/* pointers.c */

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

int main(int argc, char *argv[])  {
    int  x = 1;
    int  y = 2;
    int *p = NULL;


    printf("x = %16d, Address of x = 0x%lx, Size of x = %lu\n", x, (unsigned
long)&x, sizeof(x));
    printf("y = %16d, Address of y = 0x%lx, Size of y = %lu\n", y, (unsigned
long)&y, sizeof(y));
    printf("p = %16lx, Address of p = 0x%lx, Size of p = %lu\n\n", (unsigned
long)p, (unsigned long)&p, sizeof(p));

    puts("p = &x");
    p = &x;
    printf("x = %16d, Address of x = 0x%lx\n", x, (unsigned long)&x);
    printf("y = %16d, Address of y = 0x%lx\n", y, (unsigned long)&y);
    printf("p = %16lx, Address of p = 0x%lx\n", (unsigned long)p, (unsigned
long)&p);
    printf("*p == x ? %d\n", *p == x);
    printf("*p == y ? %d\n\n", *p == y);

    puts("p--");
    p--;
    printf("x = %16d, Address of x = 0x%lx\n", x, (unsigned long)&x);
    printf("y = %16d, Address of y = 0x%lx\n", y, (unsigned long)&y);
    printf("p = %16lx, Address of p = 0x%lx\n", (unsigned long)p, (unsigned
long)&p);
    printf("*p == x ? %d\n", *p == x);
    printf("*p == y ? %d\n\n", *p == y);

    puts("p++");
    p++;
    printf("x = %16d, Address of x = 0x%lx\n", x, (unsigned long)&x);
    printf("y = %16d, Address of y = 0x%lx\n", y, (unsigned long)&y);
```

```
    printf("p = %16lx, Address of p = 0x%lx\n", (unsigned long)p, (unsigned
long)&p);
    printf("*p == x ? %d\n", *p == x);
    printf("*p == y ? %d\n\n", *p == y);

    puts("p = &y");
    p = &y;
    printf("x = %16d, Address of x = 0x%lx\n", x, (unsigned long)&x);
    printf("y = %16d, Address of y = 0x%lx\n", y, (unsigned long)&y);
    printf("p = %16lx, Address of p = 0x%lx\n", (unsigned long)p, (unsigned
long)&p);
    printf("*p == x ? %d\n", *p == x);
    printf("*p == y ? %d\n\n", *p == y);

    return EXIT_SUCCESS;
}
```

- **REMEMBER: pointer arithmetic always has the hidden multiple of the size of the type**
  - you go down **one object not one address**