KOÇ UNIVERSITY

COMP304: PS2

Najeeb Ahmad
ENG110
nahmad16@ku.edu.tr

Koç University, Istanbul, Turkey

Agenda

- Basic C programming concepts
- Q&A Assignment 1

A C Program

- Basic Structure

```
/* Link Section */
#include <stdio.h>
/* Global variable
definition */
int tax_ID = 1;
/* Function prototypes */
float myFun(int a, float b);
/* Main function */
int main(void)
{
   printf("Hello World!\n");
   return 0;
}
```

```
/* Function definition */
float myFun(int a, float b)
{
   float result = 0;
   /* Process input */
   result = a * b;
   return
}
```

Basic I/O (stdin, stdout)

- Include stdio.h header file to access build-in functions for standard input/output
  - Standard input: keyboard
  - Standard output: screen

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

- Standard input function
  - scanf()
- Standard output function
  - printf()

Standard I/O Example

- Input age from keyboard and display on screen

```
#include <stdio.h>
int main(void)
{
    int age;
    printf("Enter your age: ");
    scanf("%d", &age);
    printf("Your age is %d". age);
    return 0;
}
```

- Format specifiers: %d, %f, %c etc.

C Pointers

- Variables that hold memory addresses.
- Syntax:

  <variable_type> *<name>;

- E.g.:

```
int *ptr_to_integer;
```

- How to get address of a variable?
  - address-of operator &

```
int x = 10;
int *ptr_to_x;
ptr_to_x = &x;
```

C Pointers

- How to get value at a memory location using pointers?
  - value-at-address operator *

```
int x = 10;
int *ptr_to_x;
ptr_to_x = &x;
printf("Value of x is %d", *ptr_to_x);
```

  - Output

```
Value of x is 10
```

Memory Allocation/Deallocation

- C provides functions to dynamically allocate/deallocate memory during runtime
  - `malloc, calloc, realloc, free`
  - Include stdlib.h to use these functions
- malloc(): Allocate specified number of bytes
  - Example

```
int *mydata;
mydata = (int *)malloc(10 * sizeof(int));
```

  - Returns NULL if unsuccessful
- calloc(): Allocates and initializes specified number of bytes to 0

Memory Allocation/Deallocation

- realloc(): Resizes a block of memory

```
mydata = (int *)realloc(mydata, 20 * sizeof(int));
```

- free(): Deallocates block of memory

```
int *mydata;
mydata = (int *)malloc(10 * sizeof(int));
/* Calculations on mydata */
…
free(mydata);
```

C Strings

- One-dimensional array of characters terminated by null character '\0'
- Defining strings

```
/* Example 1 */
char name[50];

/* Example 2
char *desig = "General Manager"

/* Example 3 */
char *address;
address = (char *)malloc(256 * sizeof(char));
```

C Strings

- String input
  - Simple solution is to use `fgets`
  - `fgets` can input from stdin or file
  - Example:

```
char address[256];
printf("Enter your address: ");
fgets(address, 256, stdin);
printf("Your address is %s", address);
```

String manipulation

- `string.h` contains string manipulation functions
  - strcmp(): Compares two strings for equality

```
int strcmp(char *s1, char *s2)
```

    - returns 0 if equal
    - returns negative if s1 is less than s2
    - returns positive if s1 is greater than s2
  - Example:

```
char name[50];
fgets(name, 50, stdin);
if(strcmp(name, "admin") == 0)
    printf("You are the administrator");
```

String manipulation

- strcpy(): Copies one string to another

```
char * strcpy(char *dest, char *src);
```

- strcat(): Concatenate two strings

```
char * strcat(char *dest, char *src);
```

  - src is appended at the end of dest.

- strlen(): returns length of a string (doesn't include null character)

```
size_t strlen (const char *s);
```

Multidimensional Arrays

- One-dimensional array

```
<type> name[size];
```

  - Example

```
int scores[100];
```

- Multidimensional array

```
<type> name[size1][size2] …. [sizen]
```

  - Two-dimensional example

```
float Amat[50][50];
```

C Structures

- Structures allow construction of user defined data types. Allow grouping of many types into a single one.
  - Declaration

```
struct typeName
{
    <type> varname1;
    <type> varname2;
    ...
} ;
```

C Structures

- Instantiation

```
struct typeName mystruct;
```

- Accessing members
  - Through object

```
mystruct.varName1
mystruct.varName2
```

  - Through pointer object

```
struct typeName *ptrtostruct;
ptrtostruct = &mystruct;
ptrtostruct->varName1
ptrtostruct->varName1
```

C Structure

- fileinfo304 struct from P4 in A1

```
struct fileinfo304
{
    size_t size;
    char filename[100];
    char datecreated[100];
    int owner_id;
    int file_id;
    struct list_head list;
}
```

- struct `list_head` is defined in linux/types.h

Linux linked list

- Linux kernel provides circular doubly linked list to the developers
- Normally linked lists are made up of structures
- Include `list_head` in the structure to use linux linked list functions
  - Requires `linux/list.h` and `linux/types.h`

Linux linked list

- Using linked list
  - Define pointer to your data structure
  - Allocate memory for data structure using `kmalloc`
  - Populate structure elements
  - Initialize list head using INIT_LIST_HEAD
  - Use list_add or list_add_tail functions to add entries to the list
  - Use traversal functions to traverse linked list
  - Useful link:

http://www.makelinux.net/ldd3/chp-11-sect-5

Ordinary Pipes

- Allow process communication in producer-consumer fashion
  - Producer process writes to one end of pipe
  - Consumer process reads from the other end
- Using pipes
  - Define integer array of size 2 as file descriptor

```
int fd[2];
```

  - Construct pipe using `pipe` function (defined in `unistd.h`)
    - returns -1 if unsuccessful

```
pipe(fd);
```

Ordinary Pipes

  - fd[0] refers to read end, fd[1] to write end
  - If process will write to the pipe, close reading end and vice versa
  - Write to the pipe using `write` function and fd[1]
  - Read from the pipe using `read` function and fd[0]
- Ordinary pipe can be accessed by process who created it and its children processes
- Ordinary pipe example in the book

QUESTIONS