Short Course on Programming in C/C++

Organized by Onur Pekcan

Contributor Selim Temizer Instructor Hasan Yılmaz

Week 2 – Lecture1

Today

We will cover;

- **Structures and Unions**
  - Basic of Structures
  - Structures and Functions
  - Structures and Arays
  - Self-Referential Structures(Structures Containing Pointers)
  - Unions & Enumerations
- **File Processing with C**
  - Reading from & Writing to Files

Basics of Structures

Structures and why we need them

```
struct person{
        int age;
        char gender;
        char * name;
};
struct person ali =
        {10, 'm', "Ali Veli"};
```

- Why do we need them?
  1. Grouping
  2. Modularity
  3. Flexibility
  4. …

```
printf("%d %c %s\n", ali.age, ali.gender, ali.name);
```

Basics of Structures

Syntax of Structures

➢ Definition

struct <type_label> {
<var_type> <var_name>;
<var_type> <var_name>;
….
<var_type> <var_name>;
<var_type> <var_name>;
};

Usage

- **struct** new_str <var_name1>, <var_name2>, …;
- Intialization:
  - **struct** new_str <var_name1> = {value1, value2, …, valueN};
- Individual Elements:
  - new_str.<var_name> = <value>
- Basically, you can use members of a struct like a variable.

Basics of Structures

Initialization of Structures

- struct student {

```
        int age;
        char gender;
        char * name;
        int grades[3];
};
```

struct student ali =

{21, ‘m’, “Ali Veli”, {60, 70, 80}};

struct student veli = {21};

➔ initializes the rest of the members to zero.

Basics of Structures

Typing and Assignment of structs

- The following are two different data types for C:

```
struct {char a; int b;} var1;
struct {char a; int b;} var2;
struct str1 { char a; int b; } var3;
struct str2 { char a; int b; } var4;
```

- For the first and the second cases, since no explicit name was given to the structure, we can't declare a new variable that has the same type as var1 and var2.

Basics of Structures

Typing and Assignment of structs

- You can assign structs of the same type to each other. E.g.:

```
struct str_type {char a; int b;};
struct str_type a = {‘m’, 10};
struct str_type b = a;
```

Basics of Structures

The "." (dot) operator

- For accessing the members of a structure, we use the dot operator:

```
struct str_type {char a; int b;} var1;
var1.a = 'm';
```

- The dot operator has the same precedence with [], & and ->
- These operators are left-to-right associative.

Basics of Structures

Size of a struct

- You can use the sizeof operator on structures.
- The size of a struct may be more than the sum of the sizes of its members.
- For example:

  struct str_type {char a; int b;} var1;

  ➔ The size of var1 is probably more than 5 (due to **data alignment** with memory words)

- However, the following is probably 2 times the size of an int:

  struct str_type2 {int a; int b;} var2;

Basics of Structures

Nested structures

- You can use one struct within another one:

```
struct name_str {
        char * first_name;
        char * last_name;
};
struct person {
        struct name_str name;
        int age;
        char gender;
}
```

```
struct person ali =
    {{“ali”, “veli”}, 10, ‘m’};

struct name_str name =
    {“veli”, “deli”};
ali.name = name;
ali.name.first_name =
        “Deli”
```

Basics of Structures

Structure Pointers

```
struct person ali =
    {{“ali”, “veli”}, 10, ‘m’};
struct person *
person_ptr;
person_ptr = &ali;

(*person_ptr).age = 20;
```

➔ person_ptr->age = 20;

- Using pointers to structures is better/faster than using structures directly especially in the case of function calls.

Basics of Structures

typedef

```
typedef struct struct_name
{
        /* variables */
} struct_name_t;


struct_name_t struct_name_t_instance;
```

Example

```
#include <stdio.h>

struct database {
   int id_number;
   int age;
   float salary;
 };

int main()
{
  struct database employee;  /*There is now an
                               employee variable that
                               has modifiable*/
                  // variables inside it.
  employee.age = 22;
  employee.id_number = 1;
  employee.salary = 12000.21;
}
```

```
#include <stdio.h>

struct database {
   int id_number;
   int age;
   float salary;
 };

int main()
{
  struct database *employee;  /*There is now an
                               employee variable that
                               points a structure*/
                  // variables inside where it points.
  employee->age = 22;
  employee->id_number = 1;
  employee->salary = 12000.21;
}
```

Example

```
#include <stdio.h>

struct student {
   int id;
   char *name;
   float percentage;
   } student1, student2, student3;

int main() {

  struct student st;

  student1.id=1;
  student2.name = "Angelina";
  student3.percentage = 90.5;

  printf(" Id is: %d \n", student1.id);
  printf(" Name is: %s \n", student2.name);
  printf(" Percentage is: %f \n", student3.percentage);

return 0;
}
```

Output:
Id is: 1
Name is: Angelina
Percentage is: 90.500000

Structures & Functions

- You can define a new structure within a function.
- In that case, the definition of that structure is accessible only within that function.

- You can pass structures as parameters to a function.
- A function can return a structure as its value.
- Since call-by-value means copying the members of structures, pointers are preferred as function parameters for structures.

Structures & Functions

- What is wrong with the following?

```
struct str {int a; char b;};
struct str * f()
{
      struct str a;

return &a;
}
```

Correct way:

```
struct str {int a; char b;};
struct str * f()
{
   struct str * a =
      (struct str *)
           malloc(
             sizeof(struct str) );
return a;
}
```

Structures and Arrays

Arrays of structures

```
struct student {
        int age;
        char * name;
        int grades[3];
};

struct student
  shortc_students[20];
```

```
struct student students[2] = {
  {10, “Ali”, {10, 20, 30},
  {20, “Veli”, {20, 30, 40}}
}
```

Self Referential Structures

Structures Containing Pointers

- Members of a structure can be pointers, as we have seen before:

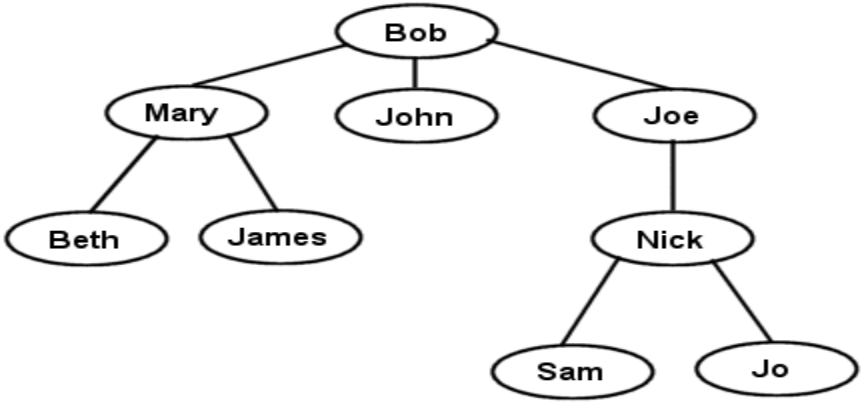
```
struct student {
        int age;
        char * name;
        int * grades;
};
```

- A structure can include a pointer to itself:

```
struct student {
    int age;
    char * name;
    struct student * friends;
    int num_of_friends;
};
```

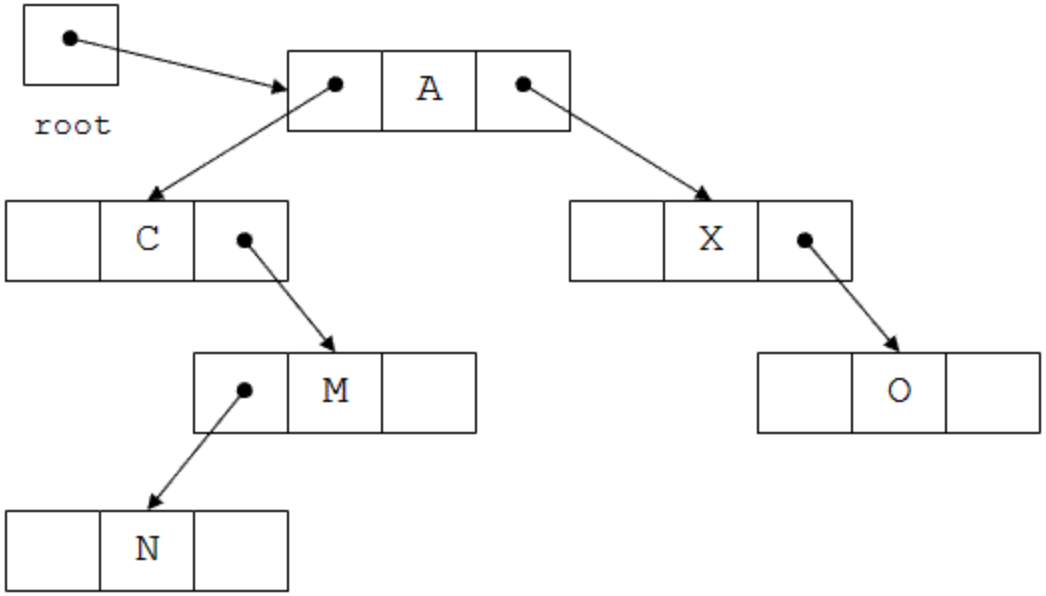
Family Tree Example with Structures

- Each person has:
  - a name, age, social security number.
  - a father and a mother.
  - siblings
  - friends
  - daughters and sons



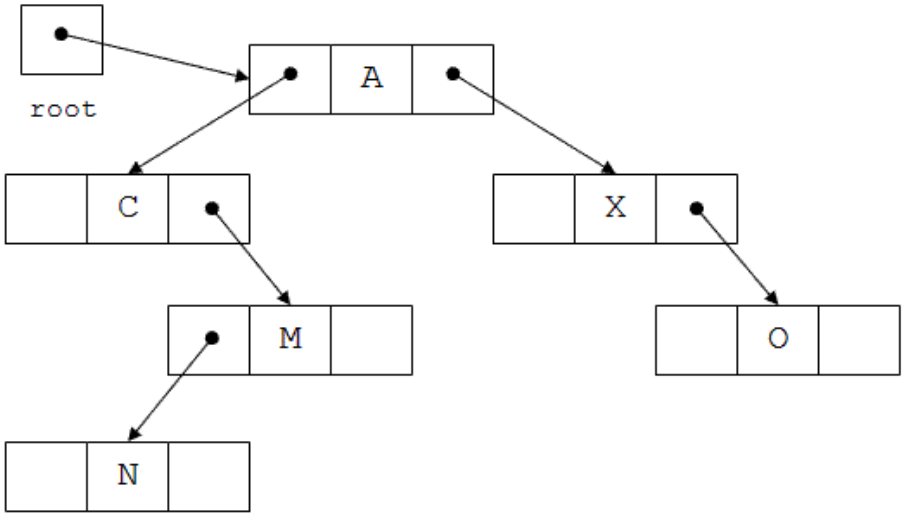
- Given such a family tree, you can implement functions to find:
  - the grandparents of a given person,
  - the cousins of a given person,
  - the grandsons of a person,
  - etc.

Structures & Pointers: Trees



Structures & Pointers: Trees

- Traversal of a tree
- Pre-order traversal:
  - A C M N X O
- In-order traversal:
  - C N M A X O
- Post-order traversal:
  - N M C O X A



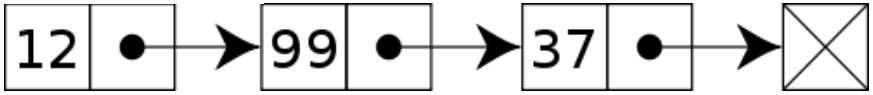
Structures & Pointers: Trees

```
void preorder_traversal(struct node * root)
{
     if( root == NULL ) return;
     printf("%c", root->value);
     preorder_traversal(root->left);
     preorder_traversal(root->right);
}
void inorder_traversal(struct node * root)
{
     if( root == NULL ) return;
     inorder_traversal(root->left);
     printf("%c", root->value);
     inorder_traversal(root->right);
}
void postorder_traversal(struct node * root)
{
     if( root == NULL ) return;
     postorder_traversal(root->left);
     postorder_traversal(root->right);
     printf("%c", root->value);
}
```

Structures & Pointers: Linked Lists

```
struct node * start;
start = make_node(12);
start->next = make_node(99);
start->next->next = make_node(37);
struct node * make_node(int value)
{
    struct node * tmp = (struct node*)
            malloc(sizeof(struct node));
    tmp->next = NULL;
    tmp->value = value;
return tmp;
}
```

```
struct node {
    int value;
    struct node * next;
    };
```

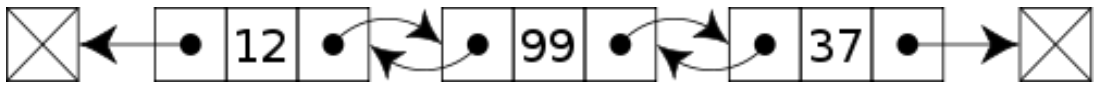


Structures & Pointers: Doubly Linked Lists

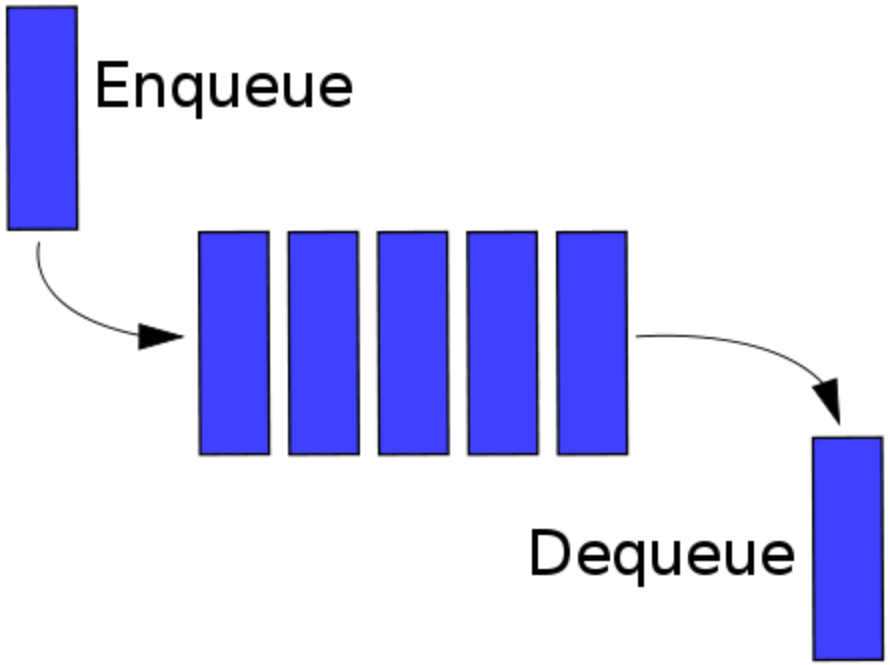
```
struct node * start;
start = make_node(12);
start->next = make_node(99);
start->next->prev = start;
start->next->next = make_node(37);
start->next->next->prev = start->next;

struct node * make_node(int value)
{
    struct node * tmp = (struct node*)
            malloc(sizeof(struct node));
    tmp->next = NULL;
    tmp->prev = NULL;
    tmp->value = value;
return tmp;
}
```

```
struct node {
    int value;
    struct node * next;
    struct node * prev;
    };
```



Application of Linked Lists: Queues



Problem 1

- Write a C program to store the information of 30 students(name, roll number and marks) using structures. Then, display the information of students

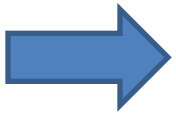
Problem 2

- Write a C program to add two distances entered by user in feet-inch system. To perform this program, create a structure containing elements feet and inch. [Note: 12 inch = 1feet]

Unions

- Allows different types of data to share the same memory location!

```
union data {
    char c;
    int i;
    double d;
    char * cp;
    };
```



- Variables c, I, d and cp share the same memory location
- The size of data is the biggest of the following:
  - sizeof(c)
  - sizeof(i)
  - sizeof(d)
  - sizeof(cp)

Unions

```
#define IS_CHAR 1
#define IS_INT 2
#define IS_DOUBLE 3
#define IS_STRING 4
union data {
     char c;
     int i;
     double d;
     char * cp;
     };
struct data_holder {
     int data_type;
     union data;
     };
```

```
struct data_holder veri;
veri.data = 20;
veri.data_type = IS_INT;

~~~~~~

if( veri.data_type == IS_INT )
     /* Do some integer comp. */
else if( veri.data_type == IS_CHAR )
     /* Do some char comp. */
/* Check for other cases */
```

Enumerations

```
#define IS_CHAR 1
#define IS_INT 2
#define IS_DOUBLE 3
#define IS_STRING 4
```

```
enum data_type
     {
      IS_CHAR, IS_INT,
      IS_DOUBLE, IS_STRING
     };

enum data_type type = IS_CHAR;
```

Enumerations

- They are essentially integers.
- The members get values starting from 0.
- Values can be assigned to the members as follows:

```
enum data_type
    {
     IS_CHAR = 2, IS_INT = 4,
     IS_DOUBLE = 6, IS_STRING = 8
    };

enum data_type type = IS_CHAR;
```

struct vs union vs enum

- What is the difference between them?
- Why do we have them?

Example

- Assume that you are given random set of characters of three types arbitrarily:
  - white space
  - alphabet letters
  - numbers
  - others
- Orn:

```
Kampusune44kis 25
   geldi universitemin?
```

- The problem is to partition these different types of data into homogeneous parts & print the partitions when requested.

File Processing with C

Files

- Files are just collections of bytes
  - One dimensional data from bytes

    01010100001110101010111010101010100001111010100 1…

- When we work with files, they are processed byte by byte.

Files & Streams

- File: a stream of bytes
  - Text stream
  - Binary stream
- Types of I/O:
  - Unbuffered
  - Fully buffered
  - Line buffered

Buffers & Buffering

- Why do we have buffers?
  - Synchronization between processes or hardware components.
    - Ex: I/O, telecommunication networks.
  - Pooling for collecting data before processing:
    - Ex: printing, online video streaming.
  - …

FILE structure

- When a file is opened, a FILE structure (called file pointer) is associated.
- FILE holds the following which are necessary for controlling a stream:
  - the current position in the file
  - error indicator
  - end-of-file indicator
  - pointer to the associated buffer
- After the processing is finished, FILE should be closed.

Standard Streams in C

- When a program starts, it is given three streams:
  - stdin: terminal keyboard or an input file
  - stdout: screen or any re-directed file
  - stderr: screen or any re-directed file
- Ex:
  - ./my_prog < in.txt 1> out.txt 2> err.txt
- Why do we have stderr?

Types of File Processing

- Sequential
  - Read the bytes in sequence
- Random Access
  - Read the bytes at a given position in the file

Opening and Closing Files in C

- FILE* fopen(const char *filename, const char * filemode)
- int fclose(FILE *filepointer)
- int fflush(FILE *filepointer)
- filemode can be:
  - “r” → Open file for reading.
  - “w” → Open file for writing. Delete old contents if it exists already.
  - “a” → Create a new file or append to the existing one.
  - “r+”, “w+”, “a+” → input & output
  - An additional “b” can be appended to the file mode for binary I/O.

Operations on Files

- int remove(const char *filename)

- int rename(
  const char *oldname,
  const char *newname)

Sequential File I/O in C

- int fscanf(FILE *fp, const char * format, …)
- int fprintf(FILE *fp, const char * format, …)

- int fgetc(FILE *fp)
- int fputc(int c, FILE *fp)

- char *fgets(char *s, int n, FILE *fp)
- char *fputs(const char *s, FILE *fp)

Random Access in C

- int fseek(FILE *fp, long offset, int whence)
- long ftell(FILE *fp)
- void rewind(FILE *fp)

- whence:
  - SEEK_SET, SEEK_END, SEEK_CUR

- size_t fread(void *s, size_t sz, size_t n, FILE *fp)
- size_t fwrite(const void *s, size_t sz, size_t n, FILE *fp)

Error Handling in File I/O

- int feof(FILE *fp)
- int ferror(FILE *fp)
- void clearerr(FILE *fp)

More on Preprocessing

- #ifdef
- #endif

Example

- Write a program that does simple encryption on a text file.