Programming in C

By

Dr Ramkumar Krishnamoorthy

kramdharma@gmail.com

Dynamic Data structures - Pointers

Unit IV

Chapter I

Contents

Pointers

- **Declaration and Initialization**
- The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer.

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

- **Syntax**
- **Datatype *varname;**
- **int *ptr;**
- **Example**
- **int n = 10;**
- **int* p = &n;**

- **Here, * refers the indication of pointer variable**
- **& refers to the address of the particular variable**

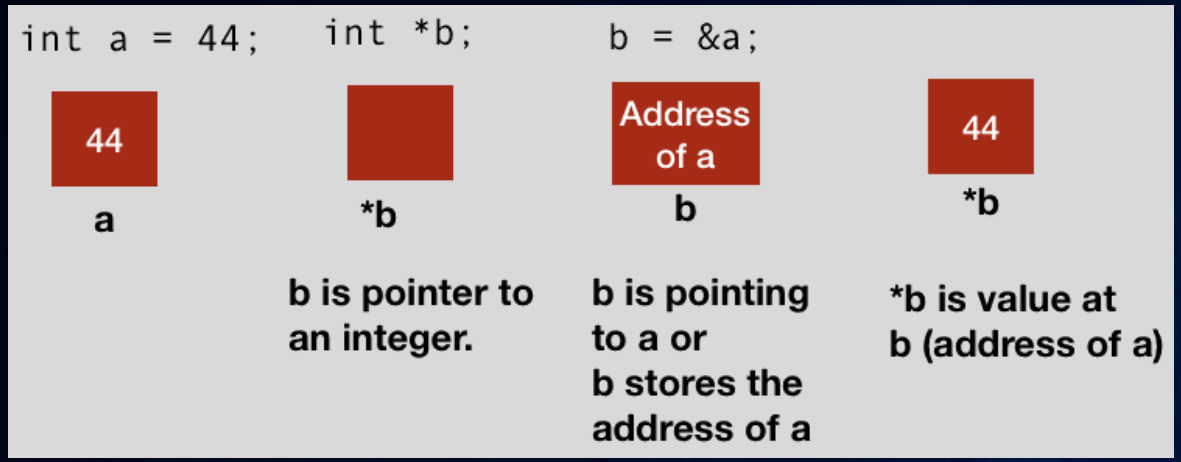
Example for Pointers

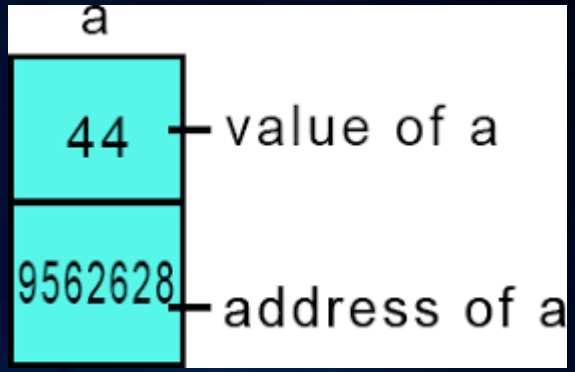
```
#include<stdio.h>
#inlcude<conio.h>
void main()
{
        int number=50;
        int *p;
        p=&number;
        printf("Address of p variable is %x \n", p);
        printf("Value of p variable is %d \n",*p);

}
```

Address of p variable is fff4
Value of p variable is 50

p contains the address of the number therefore printing p gives the address of number.

The * and & Operator





& means the address-of, you will see that in placeholders for functions to modify the parameter variable as in C, parameter variables are passed by value, using the ampersand means to pass by reference.

* means the dereference of a pointer variable, meaning to get the value of that pointer variable

Example for Without Pointers

```
#include <stdio.h>
int main ()
{
   int  var1;
   char var2[10];
   printf("Address of var1 variable: %x\n", &var1 );
   printf("Address of var2 variable: %x\n", &var2 );
   return 0;
}
```

Output

Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6

Null Pointers Example

```
#include <stdio.h>
int main ()
{
   int  *ptr = NULL;
   printf("The value of ptr is : %x\n", ptr  );
   return 0;
}
```

The value of ptr is 0

Pointer Examples

```
#include <stdio.h>
int main ()
{
   int  var;
   int  *ptr;
   var = 30;
   ptr = &var;

   printf("Value of var = %d\n", var );
   printf("Value available at the Address = %x\n", ptr );
   printf("Value  = %d\n", **ptr);
   return 0;
}
```

Output

Value of var = 30
Value available at the Address = 3EDB2A44
Value of var = 30

Pointer Example for Swapping numbers

```
#include <stdio.h>
void swap(int *a, int *b)
{
     int t;
     t   = *a;
    *a   = *b;
    *b   =  t;
}
int main()
{
    int num1,num2;
    printf("Enter value of num1: ");
    scanf("%d",&num1);
    printf("Enter value of num2: ");
    scanf("%d",&num2);
    printf("Before Swapping: num1=%d, num2=%d\n",num1,num2);
    swap(&num1,&num2);
    printf("After  Swapping: num1=%d, num2=%d\n",num1,num2);
    return 0;
}
```

Output

Enter value of num1: 100
Enter value of num2: 200
Before Swapping: num1=100, num2=200
After Swapping: num1=200, num2=100

Pointer Example for Swapping numbers

```
#include<stdio.h>
void area_peri(float, float*, float*);
int main()
{
    float radius, area, perimeter;
    printf("Enter radius of Circle\n");
    scanf("%f", &radius);
    area_peri(radius, &area, &perimeter);
    printf("\nArea of Circle = %0.2f\n", area);
    printf("Perimeter of Circle = %0.2f\n", perimeter);
    return 0;
}
void area_peri(float r, float *a, float *p)
{
    *a = 3.14 * r * r;
    *p = 2 * 3.14 * r;
}
```

Output

Enter radius of Circle: 9.8
Area of Circle = 301.57
Perimeter of Circle = 61.54

Arithmetic Pointer Examples

A pointer in c is an address, which is a numeric value.

Therefore, user can perform arithmetic operations on a pointer just as user can on a numeric value.

There are four arithmetic operators that can be used on pointers:

- **1. Pointer Increment ++**
- **2. Pointer Decrement --**
- **3. Pointer Addition +**
- **4. Pointer Subtraction –**

Pointer Increment (++) Expression

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
   int  var[] = {10, 100, 200};
   int  i, *ptr;
   ptr = var;
   for ( i = 0; i < MAX; i++)
   {
      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );
      ptr++;
   }
   return 0;
}
```

Output

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Pointer Decrement (--) Expression

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
   int  var[] = {10, 100, 200};
   int  i, *ptr;
   /* let us have array address in pointer */
   ptr = &var[MAX-1];
   for ( i = MAX; i > 0; i--)
   {
      printf("Address of var[%d] = %x\n", i-1, ptr );
      printf("Value of var[%d] = %d\n", i-1, *ptr );
      /* move to the previous location */
      ptr--;
   }
   return 0;
}
```

Output

```
Address of var[2] = bfedbcd8
Value of var[2] = 200
Address of var[1] = bfedbcd4
Value of var[1] = 100
Address of var[0] = bfedbcd0
Value of var[0] = 10
```

Pointer Addition (+) Expression

new_address= current_address + (number * size_of(data type))
32-bit int variable - it will add 2 * number
64-bit int variable - it will add 4 * number

Example

```
#include<stdio.h>
int main()
{
 int number=50;
 int *p;
 p=&number;
 printf("Address of p variable is %u \n",p);
 p=p+3;
 printf("After adding 3: Address of p variable is %u \n",p);
 return 0;
}
```

Output

Address of p variable is **3214864300**
After adding 3: Address of p variable is **3214864312**

Pointer Subtraction (-) Expression

new_address= current_address - (number * size_of(data type))
32-bit int variable - it will subtract 2 * number
64-bit int variable - it will subtract 4 * number

Example

```
#include<stdio.h>
int main()
{
int number=50;
int *p;
p=&number;
printf("Address of p variable is %u \n",p);
p=p-3;
printf("After subtracting 3: Address of p variable is %u \n",p);
return 0;
}
```

Output

Address of p variable is **3214864300**
After subtracting 3: Address of p variable is **3214864288**

Illegal Arithmetic with Pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence users must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below

- Address + Address = illegal
- Address * Address = illegal
- Address % Address = illegal
- Address / Address = illegal
- Address & Address = illegal
- Address ^ Address = illegal
- Address | Address = illegal
- ~Address = illegal

Comparison

Pointer values can be compared with one another.
Example an Integer based comparison is done.

```
#include <stdio.h>
int main()
{
        int a,b;
        int *pa,*pb;
        pa=&a; pb=&b;
        printf("Enter first integer: ");
        scanf("%d",pa);
        printf("Enter second integer: ");
        scanf("%d",pb);
        if(*pa==*pb)
                printf("Integers are equal.\n");
        else
                printf("Integers are not equal.\n");
        return 0;
}
```

Output

```
Enter first integer: 100
Enter second integer: 100
Integers are equal
```

Dynamic Memory Allocation

- The process of allocating memory at runtime is known as dynamic memory allocation
- Dynamic Memory Allocation is manual allocation and freeing of memory according to user programming needs
- Dynamic memory allocation in c language is possible by 4 functions of **stdlib.h** header file.
- **malloc()** **allocates single block of requested memory.**
- **calloc()** **allocates multiple block of requested memory.**
- **realloc()** **reallocates the memory occupied by malloc() or calloc() functions.**
- **free()** **frees the dynamically allocated memory.**

malloc()

Malloc in C

- “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size.
- It returns a pointer of type void which can be cast into a pointer of any form.
- It initializes each block with default garbage value
- malloc() allocates single block of requested memory.
- **Syntax**
- **ptr=(cast-type*) malloc (byte-size)**

Example

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory.
And, the pointer ptr holds the address of the first byte in the allocated memory.

malloc() Example

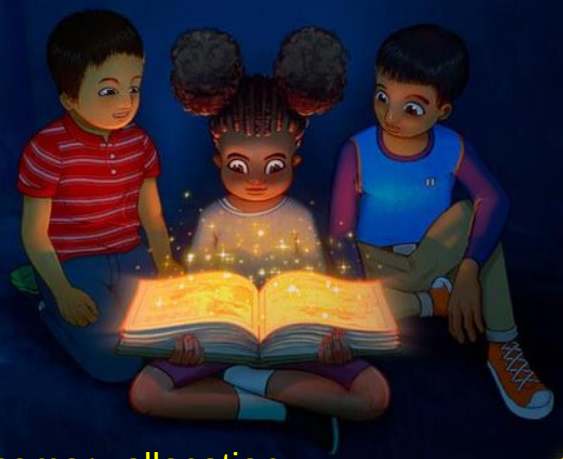
```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int *pt;
    pt = malloc (sizeof(int));
    if (pt != NULL)
    {
        printf (" Memory is created using the malloc() function ");
    }
    else
            printf (" memory is not created ");
    return 0;
}
```

Output

Memory is created using the malloc() function

calloc()

- The name "calloc" stands for contiguous allocation.
- The calloc function requires two arguments instead of one as used by malloc(). Otherwise, it operates in the same way as malloc().
- When calloc is declared for the sake of allocating a block of memory, the assigned region is immediately initialized to zero.
- It returns NULL if memory is not sufficient.
- **Syntax**
- **ptr=(data-type*)calloc(number, byte-size)**
- **Example 1:**
- **ptr = (float*) calloc(25, sizeof(float));**
- **Example 2:**
- **int*ptr;**
- **ptr = (int*) calloc(10,2); (Total 20 bytes)**
- here, 10 signifies the total number of elements that require memory allocation



malloc() vs. calloc()

malloc()	calloc()
Malloc() function will create a single block of memory of size specified by the user.	Calloc() function can assign multiple blocks of memory for a variable.
Malloc function contains garbage value.	The memory block allocated by a calloc function is always initialized to zero.
Number of argument is 1.	Number of arguments are 2.
Calloc is slower than malloc.	Malloc is faster than calloc.
It is not secure as compare to calloc.	It is secure to use compared to malloc.
Time efficiency is higher than calloc().	Time efficiency is lower than malloc().
Malloc() function returns only starting address and does not make it zero.	Before allocating the address, Calloc() function returns the starting address and make it zero.
It does not perform initialization of memory.	It performs memory initialization.

Arrays of Pointers

An array can be used with pointers to do some operations
Elements address store in consecutive locations

```c
#include <stdio.h>
const int MAX = 3;
int main ()
{
    int  var[] = {10, 100, 200};
    int i, *ptr[MAX];
    for ( i = 0; i < MAX; i++)
    {
        ptr[i] = &var[i];
    }
    for ( i = 0; i < MAX; i++)
    {
        printf("Value of var[%d] = %d\n", i, *ptr[i] );
        printf("Address of var[%d] = %x\n", i, ptr[i] );
    }
    return 0;
}
```

Output

```
Value of var[0] = 10
Address of var[0] = D27EE67C
Value of var[1] = 100
Address of var[1] = D27EE680
Value of var[2] = 200
Address of var[2] = D27EE684
```

Arrays of Pointers Example with name

```c
#include <stdio.h>
const int MAX = 4;
int main ()
{
      char *names[] = {
      "Dharma",
      "Alex",
      "Ram",
      "Mani"};

    int i = 0;
    for ( i = 0; i < MAX; i++)
    {
      printf("Value of names[%d] = %s\n", i, names[i] );
    }
   return 0;
}
```

Output

```
Value of names[0] = Dharma
Value of names[1] = Alex
Value of names[2] = Ram
Value of names[3] = Mani
```

Pointers to Pointers

- Double pointers – Adding pointers with pointers
- **1) Referencing** - Assigning an address to the pointer
- **2) Dereferencing** - Accessing the value stored at the pointer

```
#include<stdio.h>
int main()
{
    int a = 10;
    int *ptr = &a;     //ptr references a
    int **dptr = &ptr; //dptr references ptr

    printf("Address of a = %p\n",&a);
    printf("ptr is pointing to the address = %p\n",ptr);
    printf("dptr is pointing to the address = %p\n",dptr);
    printf("Value of a = %d\n",a);
    printf("*ptr = %d\n",*ptr);
    printf("**dptr = %d\n",**dptr);

    return 0;
}
```

Output

```
Address of a = 5c653f4
ptr is pointing to the address = 5c653f4
dptr is pointing to the address = 5c653e8
Value of a = 10
*ptr = 10
**dptr = 10
```

Pointers with Functions

- Pointers can be used with functions to do some tasks
- The code of a function always resides in memory, which means that the function has some address. Users can get the address of memory by using the function pointer

```
#include <stdio.h>
void main()
{
    printf("Address of main() Function is: %p", main);

}
```

Output

Address of main() Function is : 2951

Declaration of a function pointer

```
datatype (*ptr_name) (type1, type2…);
```

Example

```
int (*ip) (int);
```

In this declaration, ***ip** is a pointer that points to a function which returns an int value and accepts an integer value as an argument.

Pointers with Functions

- Example

- float (*fp) (int , int); // Declaration of a function pointer
- float func(int , int); // Declaration of function
- fp = func; // Assigning address of func to the fp pointer

```
#include <stdio.h>
int add(int,int);
int main()
{
   int a,b;
   int (*ip)(int,int);
   int result;
   printf("Enter the values of a and b : ");
   scanf("%d %d",&a,&b);
   ip=add;
   result=(*ip)(a,b);
   printf("Value after addition is : %d",result);
    return 0;
}
int add(int a,int b)
{
    int c=a+b;
    return c;
}
```

Output

```
Enter the values of a and b: 4
55
Value after addition is : 59
```

Function returning Pointers

- C language allows the users to return a pointer from a function.
- To do so, user has to declare a function returning a pointer

- Syntax
- datatype * function name(list of arguments)
- {
- set of statements;
- }
- **Example**

- int * myFunction()
- {
- set of statements;
- }

Function returning Pointers

```
#include <stdio.h>
int* findLarger(int*, int*);
void main()
{
    int a=0;
    int b=0;
    int *result;
    printf(" Input the first number : ");
    scanf("%d", &a);
    printf(" Input the second  number : ");
    scanf("%d", &b);
    result=findLarger(&a, &b);
    printf(" The number %d is larger ",*result);
}
int* findLarger(int *n1, int *n2)
{
      if(*n1 > *n2)
         return n1;
      else
         return n2;
}
```

Output

```
Input the first number : 5
Input the second  number : 6
The number 6 is larger
```

Advantages of Pointers

- Pointer reduces the code and improves the performance, it is used to retrieving strings, trees, etc.

  It can also be used with arrays, structures, and functions.
- Users can return multiple values from a function using the pointer.
- It makes user able to access any memory location in the computer's memory.

Structures

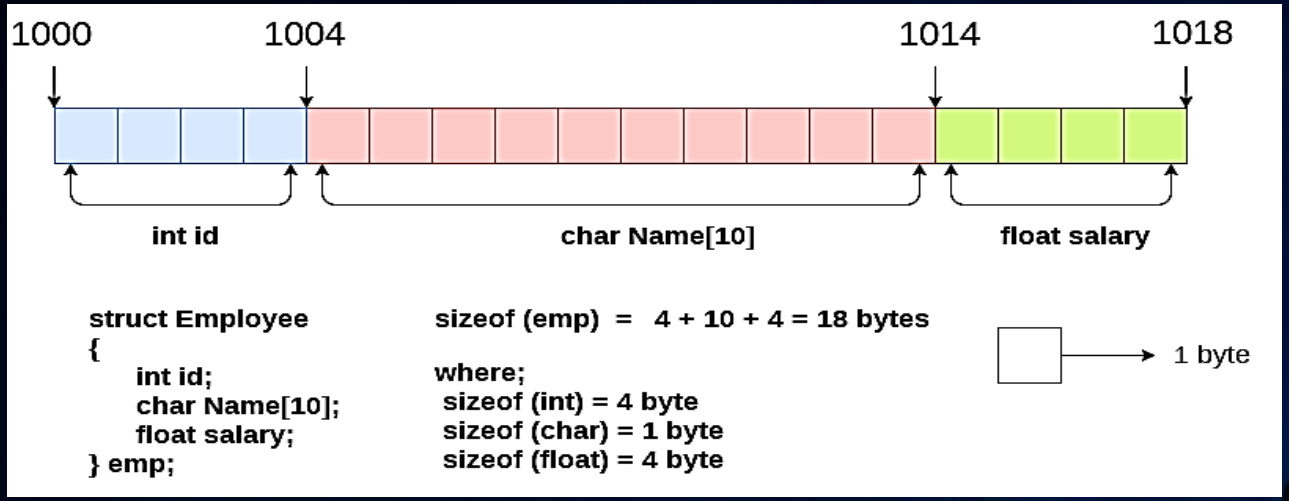
- Structure in c is a user-defined data type that enables us to store the collection of different data types.
- Each element of a structure is called a member
- Structures are used to represent a record. Suppose user wants to keep track of the books in a library.

- **Example**
  - Title
  - Author
  - Subject
  - Book ID

Syntax

```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memeberN;
};
```

Structures

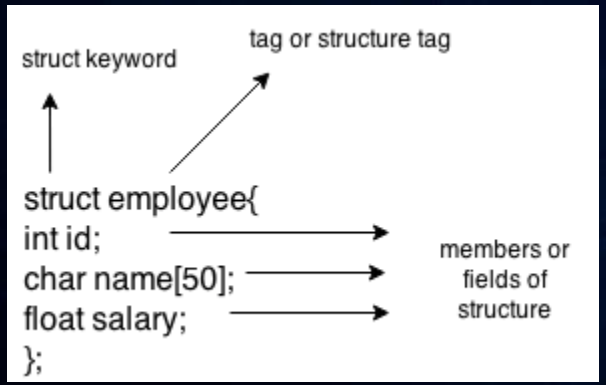



```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

sizeof (emp) = 4 + 10 + 4 = 18 bytes

where;
sizeof (int) = 4 byte
sizeof (char) = 1 byte
sizeof (float) = 4 byte

1 byte



Example

- struct employee
- { int id;
- char name[20];
- float salary;
- };

Structures

Declaring structure variable

- Users can declare a variable for the structure so that user can access the member of the structure easily.
- There are two ways to declare structure variable:
- **By struct keyword within main() function**

  By declaring a variable at the time of defining the structure

1. It should be declared within the main function.

```
struct employee
{   int id;
    char name[50];
    float salary;
};
```

struct employee e1, e2;

2. Outside main function

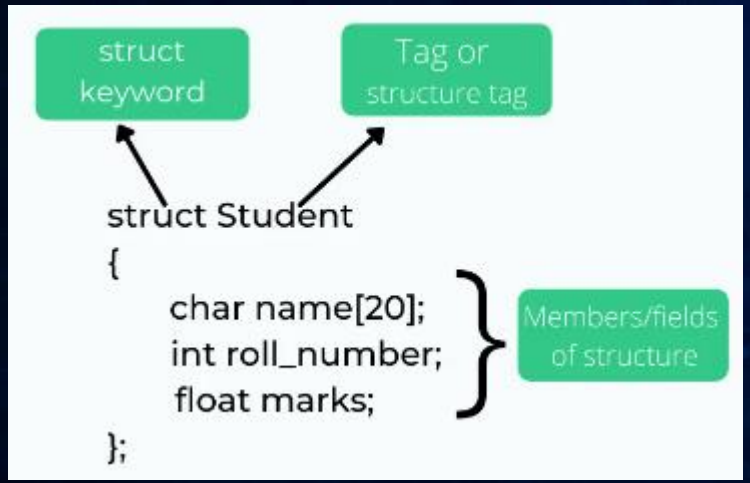
```
struct employee
{   int id;
    char name[50];
    float salary;
}e1,e2;
```

The variables e1 and e2 can be used to access the values stored in the structure

Accessing Members

Declaring structure variable

- There are two ways to access structure members:
- **By . (member or dot operator)**
- **By -> (structure pointer operator)**
- For the previous code to access the **ID member** of **p1 variable by. (member) operator**.



Accessing Members Type 1 - Example

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

struct employee
{   int id;
    char name[50];
}e1;

void main( )
{

   e1.id=101;
   strcpy(e1.name, “Alex");
   printf( "employee 1 id : %d\n", e1.id);
   printf( "employee 1 name : %s\n", e1.name);
}
```

Output

```
employee 1 id : 101
employee 1 name : Alex
```

Accessing Members Type 2- Example

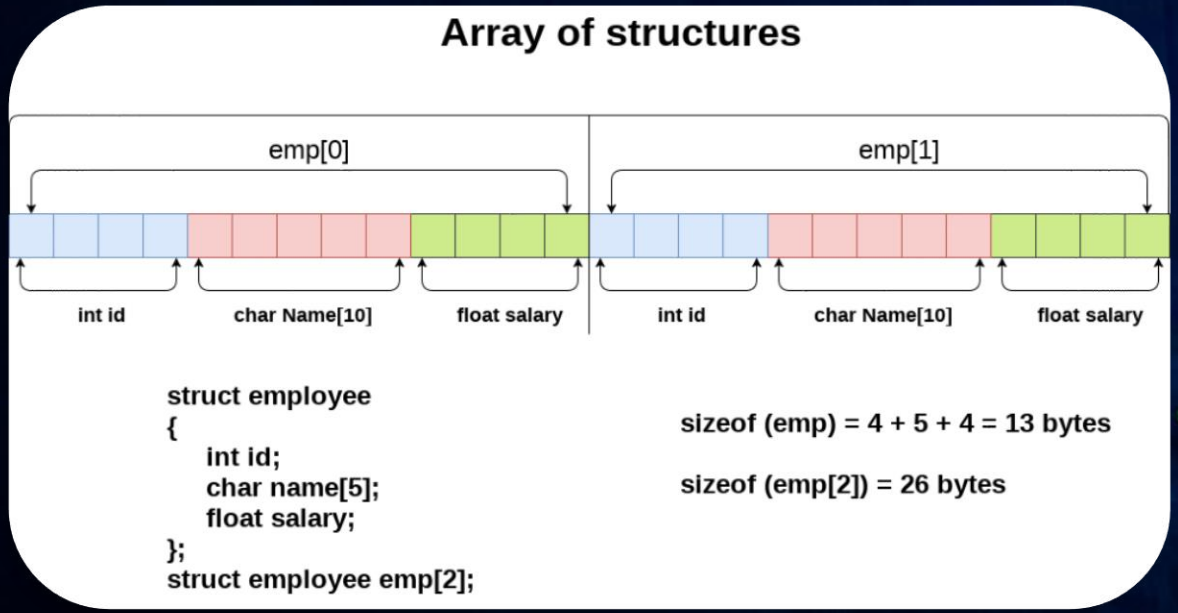
```
#include <stdio.h>
void main()
{
    struct item
    {
        int id;
        const char *company;
        float cost;
    };

    struct item radio = {101, "Sony", 99.99};
    struct item oven;
    printf("radio details: \n");
    printf("id = %d\n", radio.id);
    printf("company = %s\n", radio.company);
    printf("price = %.2f\n", radio.cost);
    oven.id = 102;
    oven.company = "Panasonic"; // assiging value using '.' operator
    printf("oven company: %s\n", oven.company);
}
```

Output

```
radio details:
id = 101
company = Sony
price = 99.99
oven company: Panasonic
```

Array of Structures

- An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities.
- The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.



Array of Structures Example

```
#include<stdio.h>
#include <string.h>
struct student
{
        int rollno;
        char name[10];
};
int main()
{
        int i;
        struct student st[5];
        printf("Enter Records of 5 students");
        for(i=0;i<5;i++){
        printf("\nEnter Rollno:");
        scanf("%d",&st[i].rollno);
        printf("\nEnter Name:");
        scanf("%s",&st[i].name);
        }
       printf("\nStudent Information List:");
       for(i=0;i<5;i++){
       printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
      }
}
```

```
Enter Records of 4 students
Enter Rollno:1
Enter Name:Dharma
Enter Rollno:2
Enter Name:Alex
Enter Rollno:3
Enter Name:Ram
Enter Rollno:4
Enter Name:Mani
```

```
Student Information List:
Rollno:1, Name:Dharma
Rollno:2, Name:Alex
Rollno:3, Name:Ram
Rollno:4, Name:Mani
```

Passing Structures to Functions

```c
#include <stdio.h>
struct student {
    char name[50];
    int per,rno;
};

void display(int a, int b);

int main() {
    struct student s1;

    printf("Enter name: ");
    gets(s1.name);
    printf("Enter the roll number: ");
    scanf("%d",&s1.rno);
    printf("Enter percentage: ");
    scanf("%d", &s1.per);
    display(s1.rno,s1.per);
    return 0;
}

void display(int a, int b ) {
    printf("\nDisplaying information\n");
    printf("Roll number: %d", a);
    printf("\nPercentage: %d", b);
}
```

Users can use the dot (.) operator to access the individual members of the structure and pass them to the function

```
Enter name: Gourav
Enter the roll number: 42
Enter percentage: 98

Displaying information
Roll number: 42
Percentage: 98
```

Structure Pointers

- Pointer to structure holds the add of the entire structure
- It is used to create complex data structures such as linked lists, trees, graphs and so on
- The members of the structure can be accessed using a special operator called as an arrow operator (->)
- **Declaration**
- Here, it is the declaration for pointers to structures in C programming −
- struct tagname *ptr; For example: **struct student *s**
- **Accessing**
- It is explained here, how to access the pointers to structures.
- Ptr**->** membername;
- For example: **s->sno, s->sname, s->marks;**

Structure Pointers Example

```
#include<stdio.h>
struct student
{
   int sno;
   char sname[30];
   float marks;
};
void main ( )
{
   struct student s;
   struct student *st;
   printf("enter sno, sname, marks:");
   scanf ("%d%s%f", & s.sno, s.sname, &s. marks);
   st = &s;
   printf ("details of the student are");
   printf ("Number = %d", st ->sno);
   printf ("name = %s", st->sname);
   printf ("marks =%f", st ->marks);
 }
```

Output

```
enter sno, sname, marks:1 Lucky 98
details of the student are:
Number = 1
name = Lucky
marks =98.000000
```

Nested Structures

- **Structures within Structures**
- Nested structures means, that one structure has another structure as member variable
- C provides users the feature of nesting one structure within another structure by using which, complex data types are created

Syntax

```
struct StructOuter
{
    structure member;
    ….

    struct  StructInner
    {
          structure member;
          ….
    };
};
```

Example

```
struct Student
{
    char[30] name;
    int age;

    struct Address
    {
        char[50] locality;
        char[50] city;
        int pincode;
    }addr;
};

struct Student Sample;
```

Nested Structures Example

```
#include <stdio.h>
#include <string.h>
struct Employee
{
        int id;
        char name[20];
    struct Date
    {
        int dd;
        int mm;
        int yyyy;
    }doj;
}e1;
```

```
int main( )
{

   e1.id=101;
   strcpy(e1.name, “Mani");
   e1.doj.dd=10;
   e1.doj.mm=11;
   e1.doj.yyyy=2014;
   printf( "employee id : %d\n", e1.id);
   printf( "employee name : %s\n", e1.name);
   printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd, e1.doj.mm,
                                        e1.doj.yyyy);
   return 0;
}
```

Output

```
employee id : 101
employee name : Mani
employee date of joining (dd/mm/yyyy) : 10/11/2014
```

Unions

- A union is a special data type available in C that allows to store different data types in the same memory location.
- Users can define a union with many members, but only one member can contain a value at any given time.
- Unions provide an efficient way of using the same memory location for multiple-purpose

Syntax

```
Union unionName
{
        datatype member1;
        datatype member2;
        ……
        datatype member1;

}
```

Example

```
union Data
{
   int i;
   float f;
   char str[20];
} data;
```

Unions - Example

```
#include <stdio.h>
#include <string.h>
union Data
{
   int i;
   float f;
   char str[20];
};
void main( )
{
   union Data data;
   printf( "Memory size occupied by data : %d\n", sizeof(data));
}
```

Output

Memory size occupied by data : 20

Unions - Example

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

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

Output

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Structure vs. Union

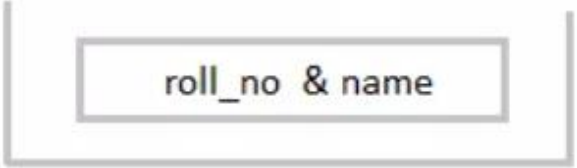
Structure

```
struct Student
{
  int roll_no;        // size 4 byte
  char name[40]; // size 40 byte
} s;
```

size of **s** will 44 bytes

Union

```
union Student
{
  int roll_no;        // size 4 byte
  char name[40]; // size 40 byte
} s;
```



size of **s** will 40 bytes

Structure vs. Union

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

Enumerated Data Type

- The **enum** is used when we want our variable to have only a set of values
- The values assigned to the enum names must be integral constant, i.e., it should not be of other types such string, float, etc
- enum names starting from 0, if users did not set values for indexing

Enumeration is a user defined datatype in C language. It is used to assign names to the integral constants which makes a program easy to read and maintain. The keyword "enum" is used to declare an enumeration.

Here is the syntax of enum in C language,

```
enum enum_name{const1, const2, ..... };
```

The enum keyword is also used to define the variables of enum type. There are two ways to define the variables of enum type as follows.

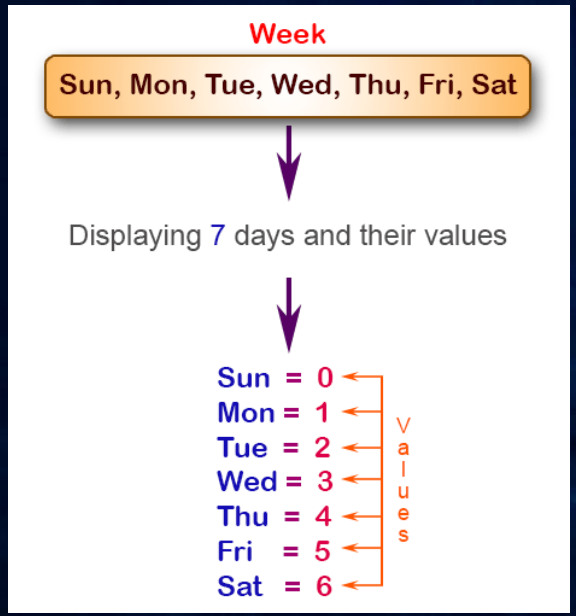
```
enum week{sunday, monday, tuesday, wednesday, thursday, friday, saturday};
enum week day;
```

Enumerated Data Type - Example

```
#include <stdio.h>
void main()
{
    enum week{Sun, Mon, Tue, Wed, Thu, Fri, Sat};
    printf("Sun = %d", Sun);
    printf("\nMon = %d", Mon);
    printf("\nTue = %d", Tue);
    printf("\nWed = %d", Wed);
    printf("\nThu = %d", Thu);
    printf("\nFri = %d", Fri);
    printf("\nSat = %d", Sat);

}
```

Output

Sun = 0
Mon = 1
Tue = 2
Wed = 3
Thu = 4
Fri = 5
Sat = 6



Enumerated Data Type - Example

```
#include <stdio.h>
enum months {jan=1, feb, march, april, may, june, july, august, september, october, november, december};
void main()
{
        for (int i=jan; i<=december; i++)
         {
                printf("%d, ",i);
         }
}
```

Output

1 2 3 4 5 6 7 8 9 10 11 12

Typedef

- The typedef is a keyword used in C programming to provide some meaningful names to the already existing variable in the C program.
- It equals the alias for the commands.
- I is used to redefine the name of an already existing variable.

Syntax

typedef **<existing_name>** **<alias_name>**

Here, '**existing_name**' is the name of an already existing variable while **'alias name'** is another name given to the existing variable

Example

```
typedef int hai;
hai i;
```

Typedef Example

```
#include <stdio.h>
void main()
{
    typedef int Fish;
    Fish i, j;
    i=10;
    j=20;
    printf("\n Value of i is :%d", i);
    printf("\n Value of j is :%d", j);

}
```

Output

Value of i is :10
Value of j is :20

```
#include<stdio.h>
void main()
{
   typedef int num;
   typedef num over_num;
   over_num a;
   printf("Enter a number: ");
   scanf("%d", &a);
   printf("You entered %d", a);
}
```

Output

Enter a number :10
You entered : 10

Typedef

```
File  Edit  Search  Run  Compile  Debug  Project  Options  Window  Help
                    \TC\TYPEDE~1.C                              4=[↕]
#include <stdio.h>
#include <conio.h>

typedef int myint;

void main()
{
int a=10;
myint b=20;
typedef myint smallint;
smallint s;
clrscr();
s=a+b;
printf("\n Sum:= %d",s);
getch();
}
                                                        17:9
F1 Help  F2 Save  F3 Open  Alt-F9 Compile  F9 Make  F10 Menu
```

alias name myint of int
this is act as integer

alias name smallint of
myint this is act as
integer

alias name s of smallint
this is act as integer

Credits

- https://www.tutorialspoint.com/cprogramming/c_pointers.htm
- https://www.javatpoint.com/malloc-in-c
- https://www.guru99.com/difference-between-malloc-and-calloc.html
- https://www.log2base2.com/C/pointer/pointer-to-pointer-in-c.html

Roadmap for Unit IV

1. Introduction to Pointers
2. Pointer Expressions and Arithmetic
3. Structures
4. Union
5. Enum and Typedef
6. Summary

Roadmap to Programming in C

Unit I
Computer Based Problem Solvingand Overview of C

1

Unit – II
Branching, Looping, Arrays, Strings

2

Unit III
Functions and Recursions

3

Unit IV
Dynamic Data Structures, Pointers, Union

4

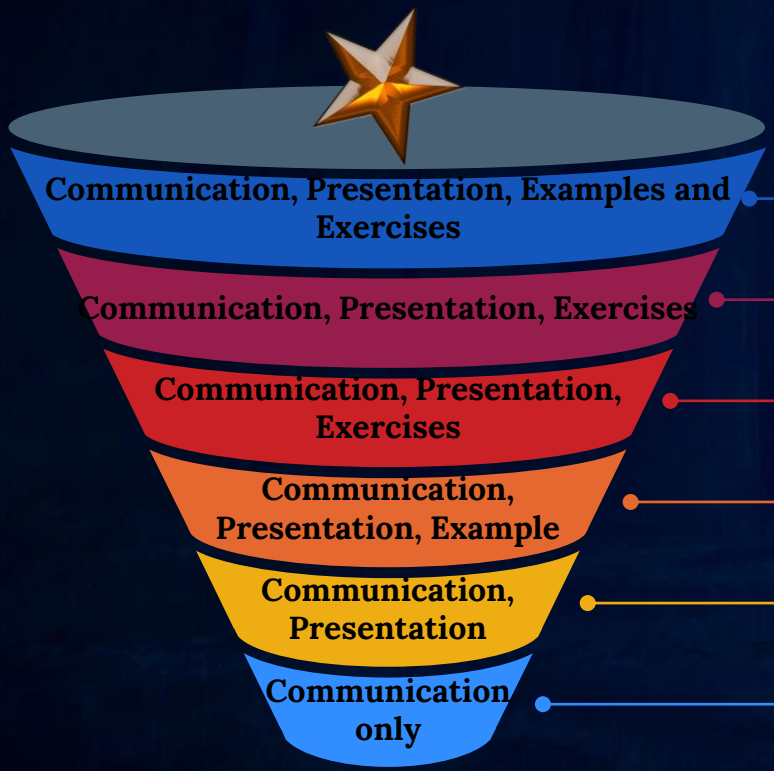
Unit V
File Handling, Preprocessor, Standard Library and Header Files

5

6

Revision

Feedback – Rating Star



$90

$80

$60

$50

$40

$30

Thanks!

Any questions?

You can find me at:

- kramdharma@gmail.com