MODULE -4

STURCUTRES AND UNIONS

Structures

Defining a Structure (structure data type)

- Arrays is one of the important data structure that permits us to group a list of data values under a single variable name called subscripted variable.
- But all the values to be stored in an array must be of **same type**.
- Hence it can't store data of different data types, so this can be done by using structures.
- **A structure** is a collection of one or more variables to store the data of different data types.
- A structure is a composite data type which contains data type which contains declaration for several items called members.
- The members of a structure can have any data type allowable in C including data type structure.
- **Structures are used to group logically related data values.**

General syntax:

```
struct struct_tag
{
     datatype1      member1;
     datatype2      member2;
     datatype3      member3;
     ……………..
     datatypeN      memberN;
} structvar1, structvar2;
```

- The word struct is a ***keyword*** in C.
- The item identified as ***struct_tag*** is called a tag or a name of the structure.
- Each ***datatype1 . . . . . data typeN***, can be of any data type in C.
- The items ***member1, member2 . . . . . memberN*** are the members of the structure.
- The items structvar1 and structvar2 are variables of the structure ***struct_tag***.
- The structure declaration does not allocate any memory or consume storage space.
- It just gives the template that conveys to the compiler the structure for the variables.

Structure variable and type definition

Ex: if we have to define a struct of a student, then related information would be:

roll_no, name, course and fees.

The structure can be declared as:

```
struct student
{
int roll_no;
char name[20];
char course[20];
float fees;
} stud1, stud2;
```

- Here we declare two variables **stud1** and **stud2** of the structure student. So if you want to declare more than one variable of the structure then separate the variables using a comma.
- When we declare a variable, memory for each variable is allocated.

  Ex2: *Declare a structure to store information of a particular date.*

```
struct date
{
int day;
int month;
int year;
};
```

Accessing members of a structure: The dot (.) operator

- The member of a structure is identified and accessed by using the dot operator.
- The dot operator connects the member name to the name of the structure.

The general form is:

```
structureName . memberName

ex:  stud1 . roll_no
     stud1 . name
     stud1 . course
     stud1 . fees
```

Structure Assignment

- ❖ It is legal to copy one entire structure to another with the same struct type by using a simple assignment statement.

Ex:

```
 struct address
{
          int houseno;
char streetno;
} homeaddress, newaddress;
.........................
Homeaddress = newaddress;
```

Initializing a structure

- It is possible to initialize a structure (or part of it) in the declaration.
- This technique is similar to initializing an array except that the members of a structure may have different data type.
- Initializing a structure means assigning some constants to the members of the structure.
- When the user does not explicitly initialize the structure, then C automatically does that.
- For **int** and **float** members the value is initialized to **zero (0).**
- For **char** and **string** the value are initialized to **NULL ('\0').**
- The initializes are enclosed in braces and are separated by commas.

The general syntax to initialize a structure variable

```
struct struct_tag
{
datatype1      member1;
datatype2      member2;
datatype3      member3;
..............
datatypeN      memberN;
} structvar = {constant1, constant2, . . . . . . .};
```

For ex: struct student

```
{
int roll_no;
char name[20];
char course[20];
float fees;
} stud1 = {01, “Puneet”, “B.E”, 45000};
```

- Here the first value is stored in first variable, second value in the second variable and so on.

```
stud1.roll_no = 01
stud1.name = Puneet
stud1.course = B.E
stud1.fees = 45000
```

Reading the values to structure

To input values for data members of the structure variable stud1, we write:

```
scanf ( “%d”, &stud1.roll_no);
scanf ( “%s”, stud1.name);
scanf ( “%s”, stud1.course);
scanf ( “%f”, &stud1.fees);
```

Displaying the values to structure

To output values for data members of the structure variable stud1, we write:

```
printff ( “%d”, stud1.roll_no);
printf ( “%s”, stud1.name);
printf ( “%s”, stud1.course);
printf ( “%f”, stud1.fees);
```

ARRAY OF STRUCTURES

- First we will analyze, where we would need array of structures.
- In a class, we do not have just one student, but there may be at least 30 students.
- So the same definition of structure can be used for all 30 students.
- This would be possible when we make an **array of structure.**
- An array of structure is declared in the same way as we had declared an array of a built in data type.

The general syntax is:

```
struct struct_tag
{
datatype1       member1;
datatype2       member2;
datatype3       member3;
……………..
datatypeN       memberN;
} ;
```

struct struct_tag   struct_var[index];

For ex:

```
struct student
{
int roll_no;
char name[20];
char course[20];
float fees;
} stud1[100];
```

- Each element of an array of structures is referenced using normal subscript notation, with the subscript right after the name of the array variable.
- To assign values to the **i**th student we write:

```
stud1[i] . roll_no = 01;
stud1[i] . name = Puneet;
stud1[i] . course = B.E;
stud1[i] . fees = 45000;
```

To output values for data members of the structure variable stud1, we write:

```
for (i = 0; i < 10; i++ )
printff ( "%d", stud1[i] . roll_no);
printf ( "%s", stud1[i] . name);
printf ( "%s", stud1[i] . course);
printf ( "%f", stud1[i] . fees);
```

STRUCTURES AND FUNCTIONS (passing structure to a function)

- A structure can be sent as a parameter to a function, it can be the value returned by a function and it can be changed in function.
- To send a structure as a parameter we use the struct type as the type of the formal parameter in the function header.
- Unlike an array, a structure is sent to a function as a value parameter.

Note :1) The position of declaration is important when you use a structure definition where it can be seen by both of them.

2) The structure type cannot be defined inside the main program; instead it must be defined above the main program so that the function can find it.

3) Any structure variable should be declared within the individual function.

For ex:
```
#include<stdio.h>
struct student
{
int roll_no;
char name[20];
char course[20];
float fees;
};
void display ( struct student);

void main ( )
{
struct student stud;
printf(“Enter the student details:”);
scanf ( “%d”, &stud. roll_no);
scanf ( “%s”, stud. name);
scanf ( “%s”, stud. course);
scanf ( “%f”, &stud. fees);
display ( stud );

getch( );
}
```

/ Function display definition */*
```
void display ( struct student stud)
{
printff ( “Roll No =%d”, stud . roll_no);
printf ( “Name = %s”, stud . name);
printf ( “Course =%s”, stud . course);
printf ( "Fees =%f", stud . fees);
}
```

Changing a structure in a Function(pointer to structure)

- If you want a function to change values in a structure, then it must receive a pointer to the structure as a parameter.
- Here we'll give the call, prototype and header for the function which receives a structure of type **struct** as a parameter, the function reads the values into the structure.
- To change a variable in a function, we send the address of that variable as the parameter by putting the **&** operator in front of the variable's name.
- If you take an example of function **readstud**to read values into the student structure, we will use the following:

  readstud (& stud);   */* function call */*

- If the actual parameter is an address the prototype and header must use ***** to indicate that the formal parameter is also an address.
- For this call to readstud, the prototype and header are the following:

```
void readstud ( struct student * );        /* function prototype */
 void readstud ( struct student * stud)
 {
              /* function definition */
 }
```

- If a pointer to a structure is passed as a parameter then, inside the function, **stud is a pointer.**
- Here we use only the arrow (→) operator for pointers.

For ex:
```
#include<stdio.h>
struct student
{
int roll_no;
char name[20];
char course[20];
float fees;
};
void readstud ( struct student * );
void printstud ( struct student  );
void main ( )
{
struct student stud;
readstud ( &stud );
printstud ( stud );
getch ( );
}
void readstud ( struct student * stud)
{
scanf ( “%d”, &stud→roll_no);
scanf ( “%s”, stud→name);
scanf ( “%s”, stud→course);
scanf ( “%f”, &stud→fees);
}
void printstud ( struct student  stud)
{
printff ( “Roll No =%d”, stud . roll_no);
printf ( “Name = %s”, stud . name);
printf ( “Course =%s”, stud . course);
printf ( “Fees =%f”, stud . fees);

}
```

Sending an Array of Structure as a Parameter

- To send an array of structures as a parameter to a function, it requires no special operators since the name of an array is an address.

```
#include<stdio.h>
struct student
{
int roll_no;
char name[20];
char course[20];
float fees;
};
void read(struct student[10] );          /*function
prototype*-/
void display (struct student[10]);

void main ( )
{

struct student stud[10];
void read(struct student stud[10] )
{
int i;
```

```
     for (i = 0; i < 10; i++ )
     scanf ( “%d”, &stud[i] . roll_no);
     scanf ( “%s”, stud[i] . name);
     scanf ( “%s”, stud[i] . course);
     scanf ( “%f”, &stud[i] . fees);
     }
     void display (struct student stud[10])
     {
     int i;
     for (i = 0; i < 10; i++ )
     printff ( “%d”, stud[i] . roll_no);
     printf ( “%s”, stud[i] . name);
     printf ( “%s”, stud[i] . course);
     printf ( “%f”, stud[i] . fees);
     }
     }
```

A Function which Returns a Structure

- A function can also return a structure by using the structure type as the return value.

```
  #include<stdio.h>
  struct student
  {
  int roll_no;
  char name[20];
  int m1, m2, m3, sum;
  float avg;
  };

/* global declaration of structure tag */
struct studmarks marks( );         /*function prototype*-/
void main ( )
{
struct studmarks     st;/* create a local structure variable */
st = marks( );                /*  function  call  which  returns  a
structure*/
printf(“ The student details :”);
printff ( “Roll No =%d”, st . roll_no);
printf ( “Name = %s”, st . name);
printf ( “Course =%s”, st . course);
printf ( “Fees =%f”, st . fees);
getch( );
}

/*function definition of marks ( ) */
struct studmarks marks( )
```

```
{
struct studmarks stu;          /*   create   a   local   structure
variable */
printf("Enter Rollno, Name, marks in 3 internals ");
scanf ( "%d%s%m1%m2%m3 ", &stu . roll_no, stu . name, &stu . m1,
&stu . m2, &stu . m3);
stu . sum = ( stu . m1 + stu . m2 + stu . m3 );
stu . avg = stu . sum / 3.0;
return stu;    /*returning a structure */

}
```

Highlights: There are 3 things to work with structure

1) One is the picture of the structure in our minds or on paper; this picture has a tree structure and shows the hierarchy & relationships among the elements.
2) The way to declare a structure, Start by defining any structure used as components. Once the structure type is defined, it can be used to declare a variable either a simple one or a component of another structure.
3) The way the elements of a structure are accessed using the dot (.) operator. To identify a member, specify each level in the structure; use the names of the variables, not the tag names of the structure types. If any level is an array, place the subscript next to the array name.

Nested Structures

- A structure can be a member of another structure. This is called nested structure.

Ex:
```
          struct student
{
int roll_no;
char name[20];
}stud;

struct studmarks
{
int m1,m2,m3;
float avg;
struct student stud;     /* structure with in structure */
}st;
```

- Here there is a structure student which is a part of structure marks.

- So in order to access structure members, use the name of the variable not the structure tag.

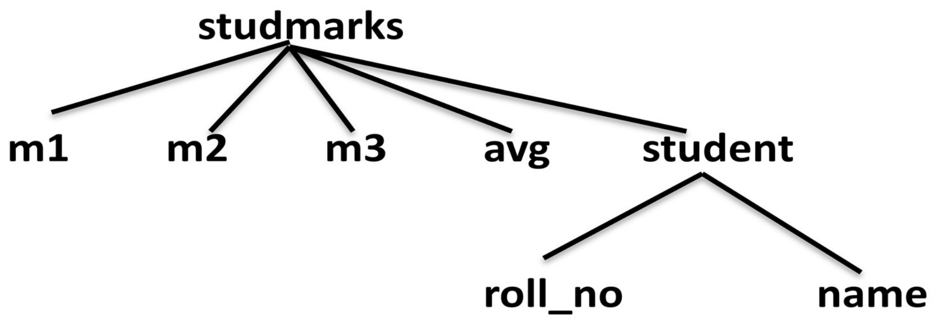
```
st . m1
st . m2
st . m3
st . avg
st . stud . roll_no
st . stud . name
```

- To read the values of the members we use:

```
          scanf ( "%d", &st . m1);
          scanf ( "%d", &st . m1);
          scanf ( "%d", &st . m1);
          scanf ( "%d", &st . stud . roll_no);
          scanf ( "%s", st . stud . name);
```

- To display the values of the members we use:

```
          printf ( "%d", st . m1);
          printf ( "%d", st . m1);
          printf ( "%d", st . m1);
          printf ( "%d", st . stud . roll_no);
          printf ( "%s", st . stud . name);
```



UNIONS

Union in C is a special data type available in C that allows storing different data types in the same memory location. You 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 purposes.

Defining a Union: To define a union, you must use the union statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows:

1.Union is like struct, except it uses less memory.

2. The keyword union is used to declare the union in C.

3. Variables inside the union are called members of the union.

```
union [union name]
    {
       member definition;
       member definition;
       ...
       member definition;
    };
```

(OR)

```
    union [union name]
    {
       member definition;
       member definition;
       ...
       member definition;
    }union variable declaration;
```

For example in the following C program, both x and y share the same location. If we change x, we can see the changes being reflected in y.

```
#include <stdio.h>

// Declaration of union is same as structures
union test {
    int x, y;
};

int main()
{
// A union variable t
    union test t;

    t.x = 2; // t.y also gets value 2
    printf("After making x = 2:\n x = %d, y = %d\n\n",
           t.x, t.y);

    t.y = 10; // t.x is also updated to 10
    printf("After making y = 10:\n x = %d, y = %d\n\n",
           t.x, t.y);
    return 0;
}
```

Output

```
After making x = 2:
 x = 2, y = 2

After making y = 10:
 x = 10, y = 10
```

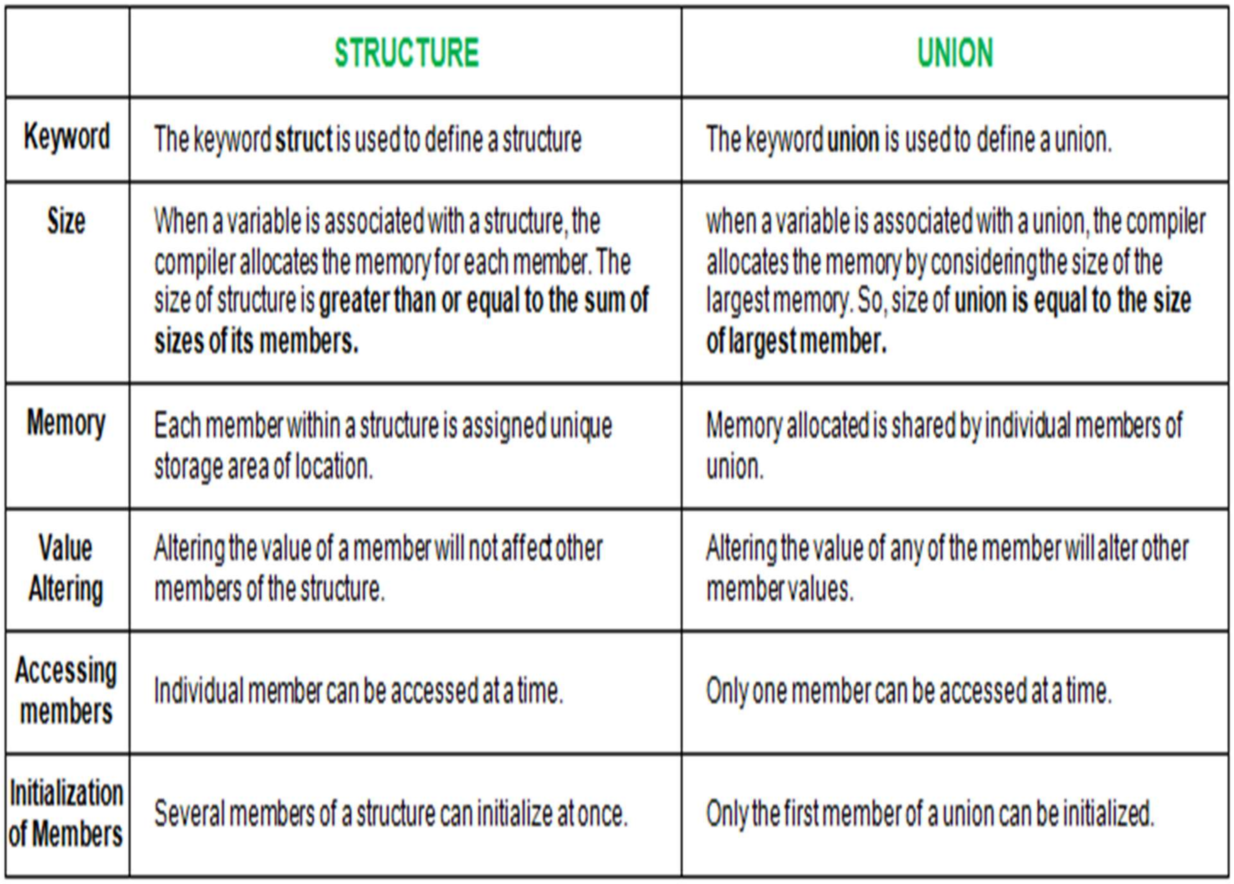
Pointers to unions?

Like structures, we can have pointers to unions and can access members using the arrow operator (->). The following example demonstrates the same.

```
#include <stdio.h>

union test {
    int x;
    char y;
};

int main()
{
    union test p1;
    p1.x = 65;

// p2 is a pointer to union p1
    union test* p2 = &p1;

// Accessing union members using pointer
    printf("%d %c", p2->x, p2->y);
    return 0;
}
```

Output:

```
65 A
```

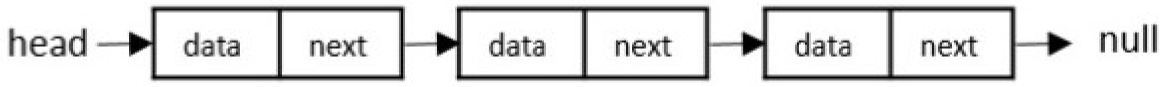


	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of 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.

Self -referential Structures

A self-referential structure is a struct data type in C, where one or more of its elements are pointer to variables of its own type. Self-referential user-defined types are of immense use in C programming. They are extensively used to build complex and dynamic data structures such as linked lists and trees.

In C programming, an array is allocated the required memory at compile-time and the array size cannot be modified during the runtime. Self-referential structures let you emulate the arrays by handling the size dynamically.



Defining a Self-referential Structure

A general syntax of defining a self-referential structure is as follows −

```
strut typename{
   type var1;
   type var2;
   ...
   ...
   struct typename *var3;
};
```

Let us understand how a self-referential structure is used, with the help of the following example. We define a struct type called mystruct. It has an integer element "a" and "b" is the pointer to mystruct type itself.

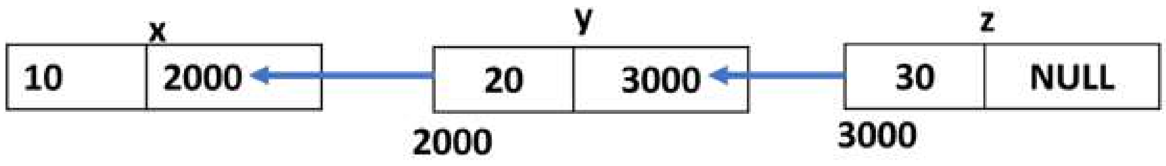
We declare three variables of mystruct type −

```
struct mystruct x = {10, NULL}, y = {20, NULL}, z = {30, NULL};
```

Next, we declare three "mystruct" pointers and assign the references **x**, **y** and **z** to them.

```
struct mystruct * p1, *p2, *p3;

  p1 = &x;

   p2 = &y;

   p3 = &z;
```

The variables "x", "y" and "z" are unrelated as they will be located at random locations, unlike the array where all its elements are in adjacent locations.



Example of Self-referential Structure

```
#include <stdio.h>

struct mystruct{

   int a;

   struct mystruct *b;

};

int main(){

   struct mystruct x = {10, NULL}, y = {20, NULL}, z = {30, 
NULL};

   struct mystruct * p1, *p2, *p3;

   p1 = &x;

   p2 = &y;

   p3 = &z;

   x.b = p2;

   y.b = p3;

   printf("Address of x: %d a: %d Address of next: %d\n", p1, 
x.a, x.b);

   printf("Address of y: %d a: %d Address of next: %d\n", p2, 
y.a, y.b);

   printf("Address of z: %d a: %d Address of next: %d\n", p3, 
z.a, z.b);

   return 0;

}
```

Output

```
Run the code and check its output −
Address of x: 659042000 a: 10 Address of next: 659042016
Address of y: 659042016 a: 20 Address of next: 659042032
Address of z: 659042032 a: 30 Address of next: 0
```

Dynamic Allocation of Memory

- ✓ In C we can allocate the memory dynamically based on the number of memory locations required.
- ✓ By doing this we overcome the ***memory wasteage***. No need of fixed memory allocation.
- ✓ When we know how many elements in an array say **"n" (**value of **'n'** as size of array is read via **scanf ( )** function**)** is required.
- ✓ Then required amount of memory can be allocated dynamically by using **pointer,** which is used to hold the address of the memory blocks.
- ✓ **Pointer** plays a vital role in **Dynamic Memory Allocation**.
- ✓ C provides built-in Dynamic memory allocation functions, they are:
  1) **malloc( )** →To allocate group of memory location called block.
  2) **calloc ( )** → To allocate multiple blocks of memory.
  3) **free ( )** → To release memory space when not required.
  4) **realloc ( )** → To alter or modify already allocated memory(to change memory size).
- ✓ These functions support Dynamic memory management, defined in **"alloc.h"** header file.

1) <u>Allocating a Block of Memory : *malloc* ( )</u>

- ✓ When we use arrays in a C program, ***static memory allocation*** takes place, for ex:

  int num[10];
- ✓ Here we can store 10 integer numbers in num array. Suppose array size increases few more integers to be read and stored or array size decreases say only 5 integers are required to be processed during execution.
- ✓ In such cases, with static memory allocation one cannot change the size of the array during program execution.
- ✓ This can be done by using **malloc ( ) which dynamically allocates the memory.**
- ✓ The function malloc ( ) returns a pointer to a void by default, the syntax is:

```
ptr = (type) malloc (size);
```

  - ➢ Where **ptr** is a pointer, which will be pointing to the memory allocated.
  - ➢ **type** is the appropriate data type.

- **malloc** is the name of the function.
- **size** is an unsigned integer representing the number of bytes that must be allocated.

Ex:

```
ptr = (int *) malloc (sizeof(n));
```

- ✓ Here **(int *)** is used to type cast the address being returned as the address of an integer.
- ✓ **ptr** is declared to be a pointer to an integer which holds starting address of **n** memory locations each of 2 bytes for storing integer numbers of an array.

2) **<u>Releasing the allocated memory : *free( )*</u>**

- ✓ It's better to vacate or clear off the dynamically allocated memory after sometime when it's no more required.
- ✓ Therefore, when already allocated dynamic memory is no longer needed, one can release it.
- ✓ This can be done by using **free ( )** function.

The general syntax is :

```
free ( ptr);
```

Here **free** is the name of the function and ptr is the address of the memory location which has to be freed or deallocated.

/* Example program illustrating **Dynamic Memory Allocation :** sum of integers */

```
#include<stdio.h>
#include<alloc.h>
void main( )
{
int i, n, sum = 0;
int *ptr;              /*pointer   to   a   memory   location
allocated dynamically */
printf(" Enter the value of n:");
scanf("%d", &n);
ptr = (int *) malloc (sizeof(n)); /* malloc( ) allocates
memory dynamically */
printf(" Enter %d integers",n);
for ( i = 0; i < n; ++i)
scanf("%d", (ptr + i) );
for( i = 0; i < n; ++i)
sum = sum + *(ptr+i);
printf("Sum of %d numbers is = %d",n, sum);
free ( ptr );              /* free ( ) releases the memory
pointed by ptr */
getch( );
}
```

Output:

```
Enter the value of n: 6
Enter 6 integers:
12
3
9
8
10
25
Sum of %d numbers is = 67
```

3) **<u>Allocating Multiple Blocks of Memory :</u>** ***<u>calloc ( )</u>***

- ✓ This function allocates multiple blocks of memory space, each of the same size and initializes all of its bytes to zero (0).[**malloc** allocates single block of memory at a time].
- ✓ Thus **calloc** is useful to allocate memory space for derived data types such as arrays and structures at runtime.
- ✓ The general syntax is:

```
ptr = (type *) calloc (n, ele-size);
```

Here **calloc** reserves contiguous memory space of “n” blocks, each of size “ele-size bytes” and returns a pointer to the first byte of the allocated memory space as required at **run time.**

- ✓ A **NULL** pointer is required if there is no sufficient memory space as required at **run time.**

Ex: A “student” structure contains data members such as rollno, name, and marks. If there are 20 students then:

```
ptr = (struct student * ) calloc ( 20, sizeof(struct student));
```

4) Altering the size of Memory : realloc ( )

- ✓ As the name indicates this function reallocates memory space with a provision for modifying already allocated memory size (**new size**).
- ✓ During reallocation one can re- correct previous allocation as to increase or decrease the memory size already allocated.
  - ➢ For ex: if the previous allocation would be:

```
ptr = malloc(size);
```

  - ➢ And now reallocation of memory space could be performed with the following statement:

```
ptr = realloc (ptr,newsize);
```

- ✓ The **realloc** allocates a new memory space of the size **newsize** and assigns it to the pointer variable **ptr** and returns a pointer to the first byte of new memory block.
- ✓ The realloc ensures that previously stored data if any, to be moved to newly reallocated memory block without any loss or data corruption.
- ✓ If sufficient memory space is not found the **realloc( )** returns **NULL**, and old block is also **lost or freed.**

Type definition *typedef*

- ➢ C allows programmers to define their own types from simpler ones.
- ➢ By using ***typedef,*** a programmer can explicitly create a new type which can be used in the program to declare variables.

The general form of a typedef definition:

```
typedef definition newtype;
```

- The word ***typedef*** is a keyword; definition represents the actual items that make up the new type, and newtype is its name.

Using typedef to Define New Types

- A common use of ***typedef*** is to define types more complicated than the basic ones available in C.
- Each definition establishes a new type which can then be used just like a predefined type such as int, char, or double.
- Using a type definition gives the programmer the same advantage that the ***#define*** directive does: a type can be used throughout the program but defined in one location.
- The programmer can associate a specific type with a specific task and more easily ensure that declarations are compatible.
- Furthermore, any changes can be made in a single location, minimizing the possibility of missing some of the references.

Examples of typedef

```
1) typedef int bigarray[1000];
   bigarray nums; is equivalent to    int nums[1000];
```

- `type bigarray represents an array of 1000 integers.`

```
2) typedef char string[80];

   string buffer;      is equivalent to    char buffer[80];
```

- `type string represents an array of 80 characters.`

```
3) typedef int *intptr;

   intptr p;           is equivalent to    int *p;
```

- `type` ***`intptr`*** `represents a pointer to an integer.`

- The first two ***typedef*** statements define types based on arrays, while the third defines one based on a pointer to an integer. Each type, once defined, is used to declare variables.

Using *typedef* to Define *struct* Types

- A more common use of typedef is to define structure types.

The general form of typedef for this purpose is shown below:

```
typedef struct tag
{
        datatype1    member1;
        datatype2    member2;
        datatype3    member3;
        ………………..
        datatypeN    memberN;
} typename;
```

- Here, ***typename*** is the name of the new ***struct*** type, and each type represents a member of the structure.
- The item ***tag*** is optional; if a name is used here, the phrase ***struct tag*** is equivalent to ***typename.***

Example:

```
typedef struct addr
{
int housenum;
char streetname[10];
} address;

address home_address, work_address;

struct addr new_address;
```

- The definition establishes a new type ***address,*** which is a structure type.
- Because ***address*** has been defined using ***typedef,*** it is used to declare variables without repeating the word ***struct.***

- Since a ***tag* – *addr*** – is supplied, we can declare variables either using type ***address*** or type ***struct addr***.

Storage Classes in C

In C, storage classes define the lifetime, scope, and visibility of variables. They specify where a variable is stored, how long its value is retained, and how it can be accessed which help us to trace the existence of a particular variable during the runtime of a program.

In C, there are **four primary storage classes:**

- auto
- register
- static
- extern

auto storage class

This is the default storage class for all the variables declared inside a function or a block. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope).

Properties of auto Variables

- **Scope:** Local
- **Default Value:** Garbage Value
- **Memory Location:** RAM
- **Lifetime:** Till the end of its scope

```
#include <stdio.h>
int main() {
      // auto is optional here, as it's the default storage
class
    auto int x = 10;
    printf("%d", x);
      return 0;
}
```

Static storage class

This storage class is used to declare static variables that have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope.

Properties of static Storage Class

- **Scope:** Local
- **Default Value:** Zero
- **Memory Location:** RAM
- **Lifetime:** Till the end of the program

```
#include <stdio.h>
void counter() {

    // Static variable retains value between calls
    static int count = 0;
    count++;
    printf("Count = %d\n", count);
}
int main() {

    // Prints: Count = 1
    counter();
      // Prints: Count = 2
    counter();
    return 0;
}
```

Register storage class

This storage class declares register variables that have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available making it much faster than any of the other variables.

Properties of register Storage Class Objects

- **Scope:** Local
- **Default Value:** Garbage Value
- **Memory Location:** Register in CPU or RAM

- **Lifetime:** Till the end of its scope

```
#include <stdio.h>

int main() {

    // Suggest to store in a register
    register int i;
    for (i = 0; i < 5; i++) {
        printf("%d ", i);
    }
    return 0;
}
```

Extern storage class

Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well.

Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block.

Properties of extern Storage Class Objects

- **Scope:** Global
- **Default Value:** Zero
- **Memory Location:** RAM
- **Lifetime:** Till the end of the program.

Examples of extern

The following examples demonstrate the use of extern keyword in C:

Access Global Variables Across Multiple Files

First create a file that contains the definition of a variable. It must be global variable,

src.c

```
// Definition of a variable
int ext_var = 22;
```

Create an extern declaration of this variable in the file where we want to use it.

main.c

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

```
// Extern decalration that will prompt compiler to
// search for the variable ext_var in other files
extern int ext_var;

void printExt() {
      printf("%d", ext_var);
}
int main() {
    printExt();
    return 0;
}
```

Assignment Questions

Remembering (Knowledge Level)

1. Define a **structure** in C and describe its syntax for declaration and initialization with an example.
2. What are **unions** in C? Explain the syntax of union declaration and initialization with an example.

Understanding (Comprehension Level)

3. Explain how **structures** are passed as parameters to functions in C with an example program.
4. Describe the concept of **self-referential structures** and illustrate with an example.
5. Differentiate between **structures and unions** in C, highlighting their similarities and differences.

Applying (Application Level)

6. Write a **C program** using structures to read, write, compute **average marks** of students, and display students scoring above and below the average for a class of 'N' students.
7. Demonstrate how to **pass a pointer to a structure** as a parameter to a function with an example program.
8. Implement a **nested structure** in C and explain its functionality with an example.

Analyzing (Analysis Level)

9. Analyze the importance of **dynamic memory allocation** in C and illustrate its implementation with a program.
10. Examine different **storage classes** in C and explain their impact on memory allocation with an example program.

************************* **End of Module -4** *******************************