DIVYA CHRISTOPHER

Functions - Module 4

C++ allows programmers to define their own functions for carrying out individual tasks. The use of programmer-defined functions allows a large program to be broken down into a number of small components, each of which has some unique purpose. A C program can be modularized through the use of functions. Use of a function avoids the need for repeated (redundant)

DOWNLOADED FROM KTUASSIST.IN

programming of the same instructions. Such programs are easy to write and debug.

A function is a self-contained program segment that carries out a well-defined task. Every C program can consists of one or more functions. Data is inputted to a function through its arguments. C's built-in function to start program execution is called main function. Same function written for a specific task, can be invoked many times from different places within a program. One function definition cannot be embedded within another function.

DOWNLOADED FROM KTUASSIST.IN

When a function is called, all the statements within that function is executed. After execution, control will be returned to the point inside main function from which the function was accessed. (i.e to the statement following the function call).

Every function returns a value using return statement, as given below:-

return expression;

Information is passed to the function via arguments or parameters.

Eg:- scanf("%d %d", &a, &b);

printf ("%d %d", a, b);

A function definition has 2 principal components:-

DOWNLOADED FROM KTUASSIST.IN

Function declaration / Function prototype

which defines the function-name along with its arguments, and function body.

Eg:- type of the function

```
data-type function-name (type1 arg1, type2 arg2 ... typeN argN)
{
    function body
}
```

(arg1, arg2 ... argN): formal arguments

Compound statement: { function body } — indicates actions to be taken up by the function.

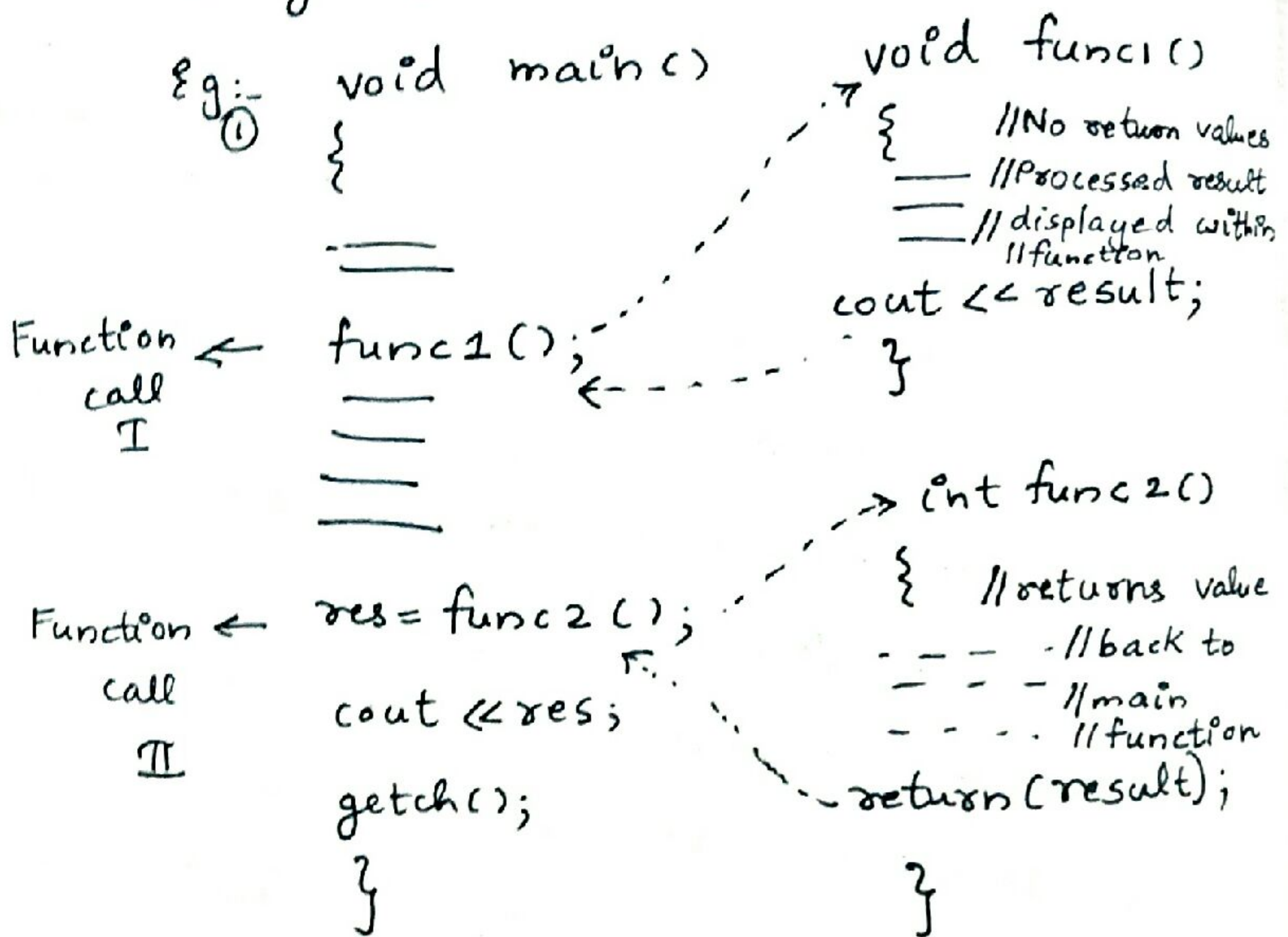
Args are separated by comma operator, and enclosed in parantheses along with and return type of the function-name and function.

The arguments in the function definition are called formal arguments.

The arguments in the function call are called actual arguments, since they define the data items that are actually transferred.

DOWNLOADED FROM KTUASSIST.IN

Through function arguments, data flows in between calling and called function. Results are transferred from user-defined function to main function using return statement.



Eg ②

```
int maximum (int x, int y)
{
  int z;
  z = (x>y) ? x : y ;
  return z;
}
```

DOWNLOADED FROM KTUASSIST.IN

Eg③

```
void maximum (int x, int y)
{
  int z;
  z = (x>y) ? x : y;
  printf("Maximum value:", z);
}
```

Eg④ :- Compute area of a circle using functions

```
#include <stdio.h>
float area (float r);
int main()
{
float rad = 10, res;
res = area(rad);
}
float area (float r)
{
float ar;
ar = 3.14 * r * r;
return (ar);
}
```

When function area( ) is invoked, value in the actual parameter is temporarily copied to formal parameters in the function definition. The resulting value after processing is returned back to main function using return statement

DOWNLOADED FROM KTUASSIST.IN

Qn) write a function to compute the power of a number?

```
#include <stdio.h>
int power(int x, int num);

int main()
{
int x=2, num=3;
int po = power (x, num);
printf ("Result is %d", po);
}
int power (int x, int num)
{
float val=1;
for(int i=1; i<=num; i++)
{
val = val * x;
}
return (val);
}
```

DOWNLOADED FROM KTUASSIST.IN

Qn) write a program to compute ^nP_r and ^nC_r of a number using functions?

$$^nP_r = \frac{n!}{(n-r)!}$$

$$^nC_r = \frac{n!}{r!\,(n-r)!}$$

```
#include <stdio.h>
int main()
{
int n, r, res1, res2;
printf(" Enter the value of n and r");
scanf("%d %d", &n, &r);
res1= fact(n)/ fact(n-r);
res2 = fact(n) / fact(r)*fact(n-r);
cout<<" npr = " << res1;
cout<<"ncr = "<<res2;
}
int fact(int n)
{
  int f=1;
for(int i=1; i<=n; i++)
  {
   f=f*i;
  }
  return (f);
}
```

DOWNLOADED FROM KTUASSIST.IN

DIVYA CHRISTOPHER

Parameter Passing Techniques

- Call-by-Value
- Call-by-Reference

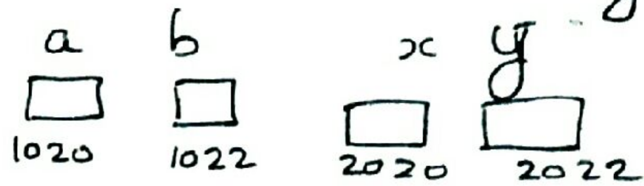
The parameters in function definition are called formal parameters and that in function call are called actual parameters.

Actual parameters are passed from calling function to formal parameters in called function and after processing values are returned back to main using return statement.

In call-by-value, data in actual parameters are temporarily copied to formal parameters in function definition. Changes made to this (formal parameters) data within callee function are not reflected back to caller, because

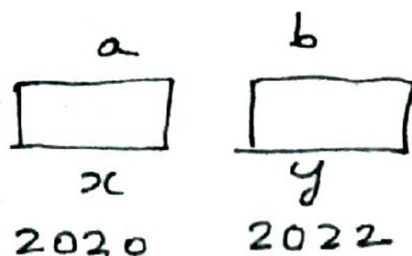
DOWNLOADED FROM KTUASSIST.IN

values are copied to different memory locations.



In call-by-reference, values are copied to same memory location and hence changes made to formal parameters data are reflected back to caller, since values are copied to same memory locations.

∴ In call by reference, Formal parameters are preceeded by & operator (address) which makes it a Reference parameter. Reference parameters directly accesses the actual arguments in the function call.



DOWNLOADED FROM KTUASSIST.IN

Eg:-
```
#include <stdio.h>
int main()
{
int x=3;
func1(x);
printf("%d", x);
func2(x);
printf("%d", x);
}
void func1(int x)
{
x=20;
}
void func2(int x)
{
x=30;
}
```

Call by Value

3	30
2502	2504

Call by Reference

30
2502

In call by value, Both of the names of formal parameters and actual parameters are independent.

DOWNLOADED FROM KTUASSIST.IN

Eg:. Swapping values of 2 variables

```
#include <stdio.h>
void swap (int , int);
void swap1 (int & , int &) ;

int main ()
{
int a=3, b=5;
printf("Original values\n");
printf ("%d \t %d", a, b);
printf(" Call by Value\n");
swap (a, b);
printf("%d \t %d", a, b);
printf("\n Call by Reference\n");
swap1 (a, b);
printf ("%d\t %d", a, b);

}
```

```
void swap(int x, int y)
{
int temp;
temp=x;
x=y;
y=temp;
}
```

```
void swap1(int &x, int &y)
{
int temp;
temp=x;
x=y;
y=temp;
}
```

In call by Reference, both the names are aliases for the same object. i.e Changes made through one is visible through the other. Thus, Actual parameters can be modified through call by reference parameter passing technique.

DOWNLOADED FROM KTUASSIST.IN

9n) Write a C program to find sum of the series $x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \ldots$

```
#include <stdio.h>
int power (int x, int y);
int fact (int N);
int main ()
{
float sum;
int term1, term2, x, count;
sum = 0.0;
printf(" Enter the value of x and N");
scanf("%d %d", &x, &N);
count = 3;
while (count <= N)
  {
    term1 = power(x, count);
    term2 = fact(count);
    sum = sum + (float) term1 / term2;
    count = count + 2;
  }
```

DOWNLOADED FROM KTUASSIST.IN

```
printf("Sum of series = %f", sum);
}
int power(int x, int y)
{
int i;
int pow=1;
for(i=1; i<=y; i++)
{
pow=pow*x;
return pow;
}
}
int fact (int N)
{
int i, fact=1;
for (i=1; i<=N; i++)
{
fact= fact*i;
}
return fact;
}
```

DOWNLOADED FROM KTUASSIST.IN

Recursion

Recursive functions are those functions that calls to itself many no: of times until terminating condition is satisfied to stop the looping action, otherwise it may result in an infinite loop.

eg:- $n! = n(n-1)(n-2)\ldots$

Factorial of a number

```
#include <stdio.h>
int fact(int);
int main()
{
 int x;
 printf("Enter the number");
 scanf("%d",&x);
```

DOWNLOADED FROM KTUASSIST.IN

```
printf("Factorial = %d", fact(x));
}
int fact(int n)
{
 int f;
 if((n==0) || (n==1))   // Terminating
                        // condition
    return 1;
 else
  f = n*fact(n-1);   // Recursive
  return f;          // function
                     // call.
}
```

O/p

Enter the number 3

Factorial = 6

DOWNLOADED FROM KTUASSIST.IN

Arrays as function parameters

Not only variables of type int, char, float and double can be passed as function arguments, but array parameters as well. When an array is passed as argument to function, only array name is passed. This array name represents the memory address of the first array element in an integer array.

DOWNLOADED FROM KTUASSIST.IN

Eg:-
```
#include<stdio.h>
#[illegible]

void disp (int marks[5]);

int main()
{
int m[]= {89, 90, 75, 70, 67};
disp(m);
}
void disp (int marks[5])
{
for (int i=0; i<5; i++)
{
printf ("Student %d", i+1);
printf ("scored %d marks", marks[i]);
}
}
```

According to eg, actual parameter m is copied to [illegible] formal parameter, int marks[5] in the function definition. This saves memory space and time.

DOWNLOADED FROM KTUASSIST.IN

Qn) Write a program to sort integer and character array using functions?

```
#include <stdio.h>
void sort1(int [], int);
void sort2(char[], int);
int main()
{
int a[10];
char b[10];
int i,n;
printf("Enter the limit");
scanf("%d", &n);
printf("Enter integer array");
for(i=0; i<n; i++)
scanf("%d", &a[i]);
printf("Enter character array");
for (i=0 ; i<n; i++)
scanf ("%d", &b[i]);
sort1(a, n);
sort2(b, n);
}
```

DOWNLOADED FROM KTUASSIST.IN

```
void sort1 (int a[], int n)
{
 int i,j,temp;
 for(i=0; i<n; i++)
 {
 for(j=i+1; j<n; j++)
  {
  if(a[i]>a[j])
   {
   temp=a[i];
   a[i]=a[j];
   a[j]=temp;
   }
  }
 }
 printf("Sorted List");
 for(i=0; i<n; i++)
 {
  printf("%d \t",a[i]);
 }
}
```

```
void sort2 (char b[], int n)
{
 int i,j;
 char temp;
 for(i=0; i<n; i++)
 {
 for(j=i+1; j<n; j++)
 {
 if(b[i]>b[j])
  {
  temp=b[i];
  b[i]=b[j];
  b[j]=temp;
  }
 }
 }
 printf("Sorted List");
 for(i=0; i<n; i++)
 {
 printf("%c\t", b[i]);
 }
}
```

DOWNLOADED FROM KTUASSIST.IN

5 advantages of using functions are :-

- Functions avoid repetition of code.
- Functions increase program readability.
- Using functions, we can divide complex problem into simpler ones.
- Functions reduce chances of error.
- Program modification is made more easier using functions.

DOWNLOADED FROM KTUASSIST.IN

Introduction to Structures

All are aware that a variable stores a single value of a datatype, while Arrays can store many values of similar datatype. In real life, we need to store different data types together, to maintain employee information comprising of name, age, qualification and salary. For tackling such mixed data types in a variable, we use structures.

A structure is a collection of one or more variables of different data types, grouped together under a single name.

DOWNLOADED FROM KTUASSIST.IN

Features:-

→ It is possible to copy the contents of all structure elements of different datatypes to another structure variable of same data type using assignment operator (=).

It is because structure elements are stored in successive memory locations.

→ Nesting of structures is possible. i.e one can create a structure within a structure. Using this feature, one can handle complex data types.

DOWNLOADED FROM KTUASSIST.IN

→ It is also possible to pass structure elements to a function, either by call-by-value or call-by-address.

→ It is also possible to create a structure pointer pointing to structure elements using member access operator (→).

→ Structure declaration starts with the struct keyword. It defines the structure, but it does not allocate memory. Memory allocation takes place only when variables are declared.

Dot operator is used when simple object is declared and arrow operator (→) is declared, when object is a pointer to a structure.

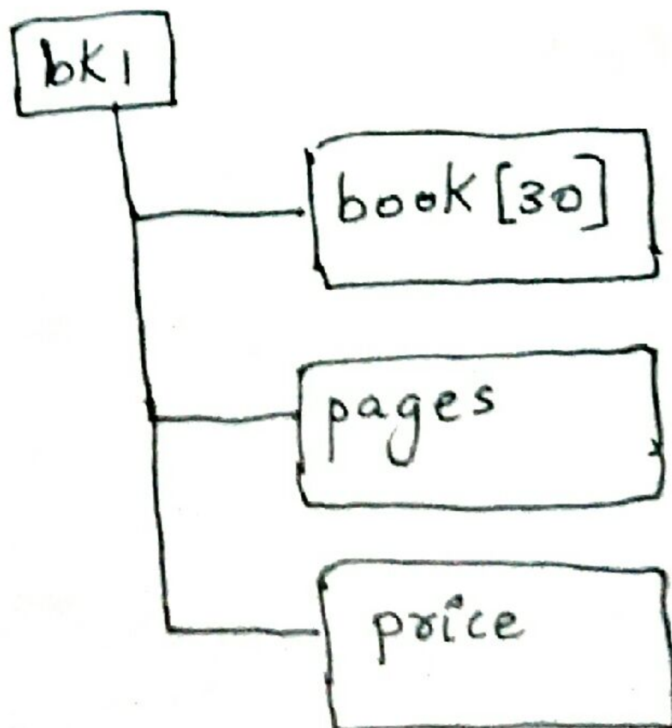
DOWNLOADED FROM KTUASSIST.IN

Eg:-

```
struct book1        //Structures
{                   //hold information
char book[30];
                    // comprising of
int pages;
                    //different
float price;
                    // data types.
}bk1;
```

A structure bk1 of type book1 is created.

Block diagram of structure.



DOWNLOADED FROM KTUASSIST.IN

Syntax

```
struct tagname
  {
    member1;   // individual
    member2;
     ....      // member declarations.
    member n;
  };
```

struct tag-name variable-name;

Structure variables of type tag are created using struct datatype. No initialization is permitted inside structure.

```
struct student
{
  char name[30];
  int rollno;
  float tot_marks;
};
struct student s;
```

s is a structure variable holding the details of structure student.

DOWNLOADED FROM KTUASSIST.IN

Structure members are accessed using structure variable as follows:-

structure variable.structure member name=value;

Eg:- s.name= "HARI"

s.rollno = 33

s.tot_marks = 92.6;

Eg:-
```
#include<stdio.h>
struct item
{
int codeno;
 float price;
 int qty;
};
int main()
{
struct item a,*b;
```

DOWNLOADED FROM KTUASSIST.IN

```
a.codeno = 123;
a.price = 150.75;
a.qty = 150;
printf("\n Code no: %d", a.codeno);
printf("\n Price: %d", a.price);
printf("\n Quantity: %d ", a.qty);
b->codeno = 124;
b->price = 200.75;
b->qty = 75;
printf("with pointer to structure");
printf("Code no: %d", b->codeno);
printf("\n Price: %f ", b->price);
printf("\n Quantity: %d", b->qty);
}
```

O/P

Codeno: 123

Price : 150.75

Quantity: 150

With pointer to structure

Codeno: 124

Price: 200.75

Quantity: 75

DOWNLOADED FROM KTUASSIST.IN

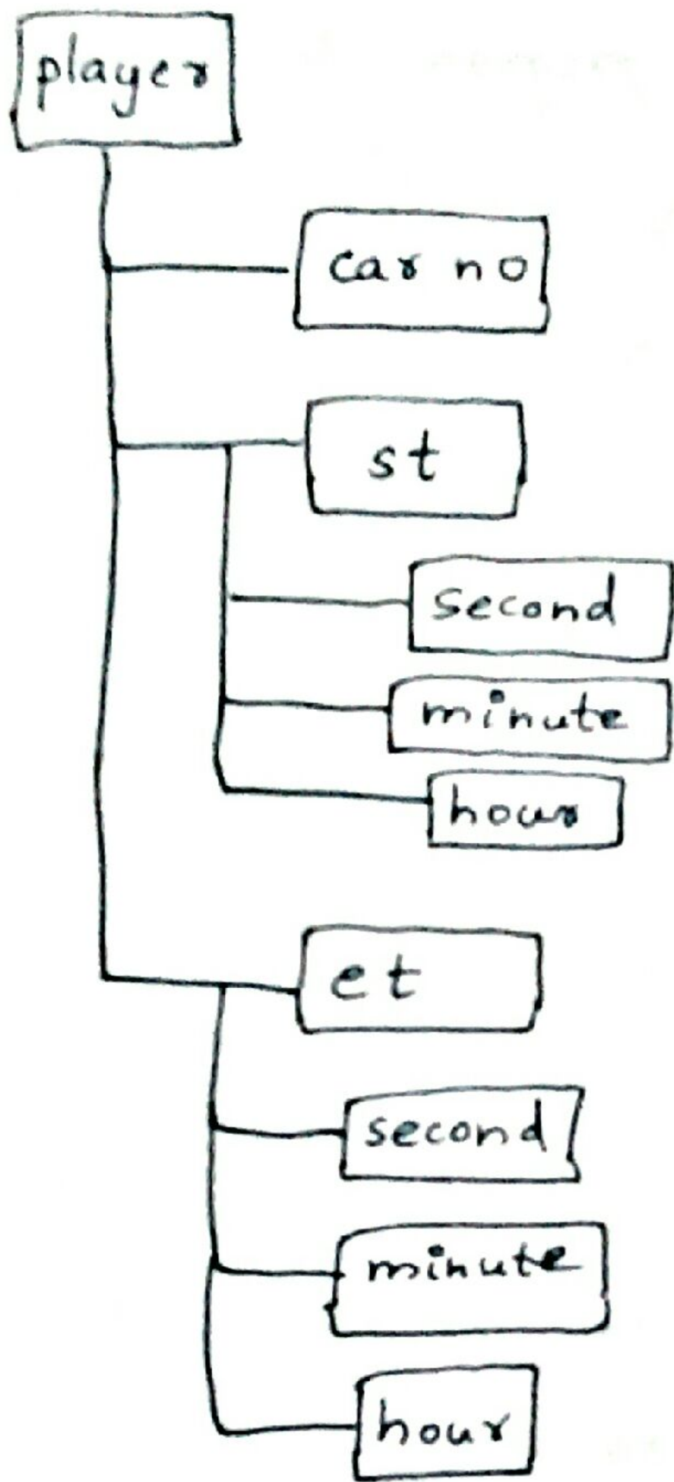
Structure within Structure (Nested Structure)

A structure variable can include another structure variable as its member.

Eg:-
```
struct time
{
 int second;
 int minute;
 int hour;
};

struct t                //structure t
{                       // contains
 int carno;             // another
 struct time st;        //structure
 struct time et;        // time as one
};                      // of its
struct t player;        // members.
```

DOWNLOADED FROM KTUASSIST.IN



Memory allocation of a nested structure

DOWNLOADED FROM KTUASSIST.IN

Qn) Write a program to enter full name and date of birth of a person? Use nested structure?

```
# include <stdio.h>
struct name
{
char first_nm[10];
char last_nm[10];
};
Struct b_date
{
int day;
int mon;
int year;
};
struct data
{
struct name nm;
struct b_date bt;
};
```

DOWNLOADED FROM KTUASSIST.IN

```
int main()
{
 struct data s;
printf ("Enter first name & last name");
 scanf ("%s %s", s.nm.first_nm,
          s.nm.last_nm);
  printf("Enter birth date");
  scanf("%d %d %d",&s.bt.day,
  &s.bt.mon,&s.bt.year);
  printf("Student details:: \n");
  printf("Name\n");
printf("%s", s.nm.first_nm, "%s", s.nm.last_nm);
  printf ("Birthdate\n");
  printf("%d / %d / %d", s.bt.day,
          s.bt.mon .s.bt.year);
   }
```

DOWNLOADED FROM KTUASSIST.IN

O/p

Enter first & lastname

Roy Jose

Enter birth date

12 10 1996

Student details.

Name:

Roy Jose

Birth date:

12/10/1996

Array of Structures

In array of structures, every array element is of structure data type. All array elements are stored in adjacent memory

DOWNLOADED FROM KTUASSIST.IN

allocations. Array of structures is a collection of array elements of similar data types, which contains dissimilar data types.

Eg:- Rank list of Students

```
#include <stdio.h>
#include <string.h>
struct student
{
char name[25];
float m1, m2, m3;
char grade[20];
};
int main()
{
struct student s[5];
struct student s1;
float tem, tot[10];
int i, n;
```

DOWNLOADED FROM KTUASSIST.IN

```
printf ("Enter the no: of students");
 scanf ("%d", &n);

for (i=0; i<n; i++)

{
printf (" Enter the name of student %d", i+1);

printf (" Enter the marks in 3 subjects");

 scanf ("%s", s[i].name);

scanf ("%d %d %d", s[i].m1, s[i].m2,
                   s[i].m3);

  tot[i]= s[i].m1 + s[i].m2 + s[i].m3;

  if (tot[i] > 250)

   strcpy (s[i].grade, "Distinction");

elseif (tot[i] >200)

   strcpy (s[i].grade, "First Class");

else if (tot[i]>120)

  strcpy (s[i].grade, "Failed");
}
```

DOWNLOADED FROM KTUASSIST.IN

// Preparation of Ranklist

```
for (i=0; i<n; i++)
{
for(int j=i+1; j<n; j++)
{
if(tot[i] < tot[j])
{
tem = tot[j];
tot[j] = tot[i];
tot[i] = tem;
s1 = s[j];
s[j] = s[i];
s[i] = s1;
}
}
}
printf("Name \t Mark1 \t Mark2 \t Mark3 \t
        Total \t Grade \t \n");

printf("- - - - - - - - - - - - - - - - - \n\n");

for(i=0; i<n; i++)
{
printf("%s \t %f \t %f \t %f \t %f \t %s \t",
s[i].name, s[i].m1, s[i].m2, s[i].m3, tot[i],
          s[i].grade);
} }
```

DOWNLOADED FROM KTUASSIST.IN

Op

Enter the no: of students 2

Enter the name of student1 Minnu

Enter the marks in 3 subjects 99 87 82

Enter the name of student2 Rose

Enter the marks in 3 subjects 30 51 44

Name	Mark1	Mark2	Mark3	Total	Grade
Minnu	99	87	82	268	Distinction
Rose	30	51	44	135	Second class

User defined data types (typedef)

It allows users to define new data types equivalent to existing data types.

A new datatype is defined as,

typedef type new-type;

DOWNLOADED FROM KTUASSIST.IN

where type refers to existing datatype and new-type refers to new user-defined data type.

Eg:-
```
typedef struct
{
    member1;
    member2;
    . . .
    . . .
    member m;
} new-type;
```

In the above eg, new type is a user-defined structure type.

Eg:-
```
typedef struct
{
    int accno;
    char acc-type;
    char name[80];
    float bal;
} record;
record oldcustomer, newcustomer;
```

DOWNLOADED FROM KTUASSIST.IN

Union

Members of a union share same storage area. Union permits several data items to be stored in the same portion of computers memory. Unions are used to conserve memory. Unions are created using union Keyword.

```
union tag
{
  member 1;
  member 2;
   . . . .
  member n;
};
```

DOWNLOADED FROM KTUASSIST.IN

union tag var-name1, ... var-namen;

Eg:- union id {

char color [12];

int size;

} shirt, blouse;

Here, we have declared 2 union variables shirt and blouse of type id.

Each individual union member can be accessed using dot operator (.) or member access operator (->).

DOWNLOADED FROM KTUASSIST.IN

Eg:-

```
#include <stdio.h>

int main ()
{
  union id {
        char color;
        int size;
      };
   struct {
     char manufacterer [20];
      float cost;
        union id description;
      } shirt, blouse

printf ("%d\n", size of (union id));
  shirt. description. color = 'w';
printf ("%c %d \n", shirt.description.color,
          shirt.description.size);
```

DOWNLOADED FROM KTUASSIST.IN

```
shirt.description.size = 12;
printf ("%c %d \n", shirt.description.color,
        shirt.description.size);
}
```

Output

2 → 2 bytes memory allocation (largest members size) int - 2 bytes

W - 24713

@ 12

Only one member of union can be assigned a value at any 1 time. Most compilers accept only an initial value for 1 union member.

DOWNLOADED FROM KTUASSIST.IN

Storage Classes in C

Storage Classes controls two different properties of a variable : lifetime and scope.

Two types of scope :- (a) local
(b) global

Local scope is limited to the function, where it is defined. The life of a local variable ends when the function exits.

Global scope covers the entire program. Global variables are defined outside all functions just below header file declaration. Its lifetime ends, only when the program ends.

DOWNLOADED FROM KTUASSIST.IN

```c
#include <stdio.h>
int c = 12; //Global variable initialization
void test();
int main()
{
int d = 5; //Local variable initialization
++d;
++c;
printf("d = %d", d);
printf("\n c= %d",c);
test();
}
void test()
{
++c;
printf("\n c= %d", c);
}
```

Output

d = 6

c = 13

c = 14

DOWNLOADED FROM KTUASSIST.IN

Storage Classes

The 4 storage classes in C are auto, static, register and extern.

(a) auto -

Automatic variables are declared using keyword auto. However, if a variable is declared without any keyword inside a function, it is automatic by default. auto storage class applies to local variables only. This automatic variable is visible only within the function where it is declared and its lifetime is the same as the lifetime of the function as well. Once the execution

DOWNLOADED FROM KTUASSIST.IN

of function is finished, variable is destroyed.

Syntax of automatic storage class declaration -

auto data-type var-name = value;

Eg:- auto int x = 2;

(b) Static -

Static local variables exists only inside a function, where it is declared and defined.

Its lifetime starts when the function is called and ends only after the program execution has finished. When a function is called, static variable defined

DOWNLOADED FROM KTUASSIST.IN

inside the function, retains its previous value and operates on it. The default initial value of a static variable is 0.

Syntax of static storage class declaration -

static datatype variablename=value;

static int x = 100;

Eg:-

```
#include <stdio.h>
int main()
{
test();
test();
}

void test()
{
static int var=0;
++var;
printf("%d",var);
}
```

Output

1
2

DOWNLOADED FROM KTUASSIST.IN

(c) register -

register storage class assigns a variables storage in the CPU registers rather than the primary memory. register variables has its lifetime and visibility same as automatic variables. The purpose of creating register variable is to increase the access speed and to make the program run faster. If there is no space available in the register, these variables occupy space in the main memory and

DOWNLOADED FROM KTUASSIST.IN

act similar to automatic storage class variables. Only those variables which require faster access is made as register.

Syntax of Register storage class declaration -

```
register datatype variable-name
                        = value;
```

Eg:- register int d;

(d) <u>extern</u> -

External storage class assigns the variable with a reference to the global variable declared outside the given program.

DOWNLOADED FROM KTUASSIST.IN

extern Keyword is used to declare external variables. These variables are visible throughout the program and its lifetime is same as the lifetime of the program where it is declared.

Syntax of external storage class declaration -

extern datatype var-name = value;

Eg :- extern int test;

Storage Class	Keyword	Life time	Visibility	Initial value
Automatic	auto	Function block	Local	Garbage value
External	extern	Whole program	Global	Zero
Static	static	Whole program	Local	zero
Register	register	Function block	Local	Garbage value

DOWNLOADED FROM KTUASSIST.IN

Eg:- File sub.c

```
int test = 100;
void multiply (int n)
 {
 test = test * n;
 }
```

File : main.c

```
#include <stdio.h>
#include "sub.c"
extern int test;
int main()
{
printf("%d", test);
multiply(5);
printf("\n%d", test);
}
```

Output

100

500

Variable test is declared as external in main.c. This variable can be accessed from both programs. The program performs multiplication and changes global variable test to 500.

DOWNLOADED FROM KTUASSIST.IN

DIVYA CHRISTOPHER

Functions - Module 4

C++ allows programmers to define their own functions for carrying out individual tasks. The use of programmer-defined functions allows a large program to be broken down into a number of small components, each of which has some unique purpose. A C program can be modularized through the use of functions. Use of a function avoids the need for repeated (redundant)

DOWNLOADED FROM KTUASSIST.IN

programming of the same instructions.

Such programs are easy to write and debug.

A function is a self-contained program segment that carries out a well-defined task. Every C program can consists of one or more functions. Data is inputted to a function through its arguments. C's built-in function to start program execution is called main function. Same function written for a specific task, can be invoked many times from different places within a program. One function definition cannot be embedded within another function.

DOWNLOADED FROM KTUASSIST.IN

When a function is called, all the statements within that function is executed. After execution, control will be returned to the point inside main function from which the function was accessed. (i.e to the statement following the function call).

Every function returns a value using return statement, as given below.

return expression;

Information is passed to the function via arguments or parameters.

Eg:- scanf("%d %d", &a, &b);

printf ("%d %d", a, b);

A function definition has 2 principal components:-

DOWNLOADED FROM KTUASSIST.IN

Function declaration / Function prototype

which defines the function-name along with its arguments, and function body. semi

Eg:- type of the function

```
data-type function-name(arg1, arg2 ... argN)      type1 type2 typeN
                                                  formal arguments
{
    function body    } indicates actions to be taken up by the function.
}
```

Compound statement

Args are separated by comma operator, and enclosed in parantheses along with function-name and return type of the function.

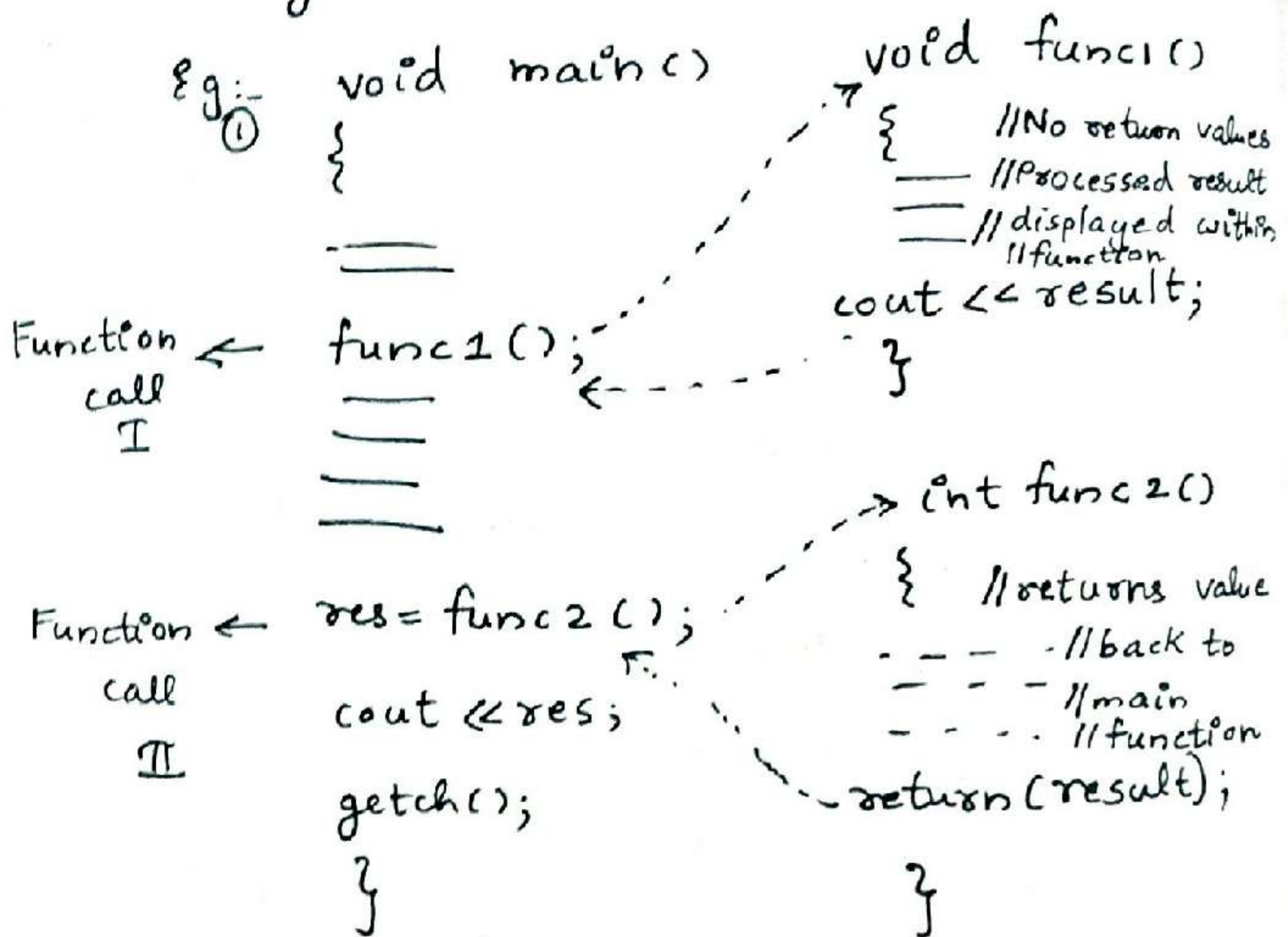
The arguments in the function definition are called formal arguments.
The arguments in the function call are called actual arguments, since they define the data items that are actually transferred.

DOWNLOADED FROM KTUASSIST.IN

Through function arguments, data flows in between calling and called function.

Results are transferred from user-defined function to main function using return statement.

Eg:- ①



Eg②
```
int maximum (int x, int y)
{
 int z;
 z = (x>y) ? x : y ;
 return z;
}
```

DOWNLOADED FROM KTUASSIST.IN

Eg③ void maximum (int x, int y)

```
{
 int z;
 z = (x > y) ? x : y;
 printf("Maximum value:", z);
}
```

Eg④ :- Compute area of a circle using functions

```
#include <stdio.h>
float area (float r);
int main()
{
float rad = 10 , res;
res = area(rad);
}
float area (float r)
{
float ar;
ar = 3.14 * r * r;
return (ar);
}
```

When function area( ) is invoked, value in the actual parameter is temporarily copied to formal parameters in the function definition. The resulting value after processing is returned back to main function using return statement

DOWNLOADED FROM KTUASSIST.IN

Qn) write a function to compute the power of a number?

```
#include <stdio.h>
int power(int x, int num);
int main()
{
int x=2, num=3;
int po = power (x,num);
printf ("Result is %d", po);
}
int power (int x, int num)
{
 float val=1;
 for(int i=1; i<=num; i++)
 {
  val = val * x;
 }
 return (val);
}
```

DOWNLOADED FROM KTUASSIST.IN

Qn) write a program to compute ^nP_r and ^nC_r of a number using functions?

$$^nP_r = \frac{n!}{(n-r)!}$$

$$^nC_r = \frac{n!}{r!(n-r)!}$$

```
#include <stdio.h>
int main()
{
int n, r, res1, res2;
printf(" Enter the value of n and r");
scanf("%d %d", &n, &r);
res1= fact(n)/ fact(n-r);
res2 = fact(n) / fact(r)*fact(n-r);
cout<<"npr = "<<res1;
cout<<"ncr = "<<res2;
}
int fact(int n)
{
 int f=1;
for(int i=1; i<=n; i++)
  {
   f=f*i;
  }
  return(f);
}
```

DOWNLOADED FROM KTUASSIST.IN

DIVYA CHRISTOPHER

Parameter Passing Techniques

- Call-by-Value
- Call-by-Reference

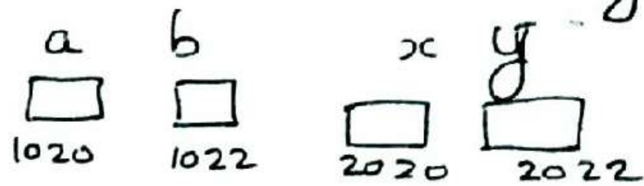
The parameters in function definition are called formal parameters and that in function call are called actual parameters.

Actual parameters are passed from calling function to formal parameters in called function and after processing values are returned back to main using return statement.

In call-by-value, data in actual parameters are temporarily copied to formal parameters in function definition. Changes made to this (formal parameters) data within callee function are not reflected back to caller, because

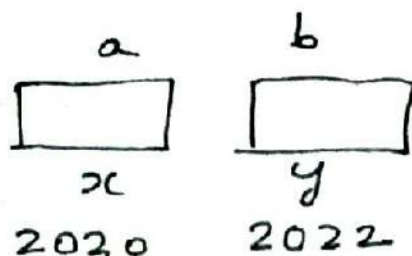
DOWNLOADED FROM KTUASSIST.IN

values are copied to different memory locations.



In call-by-reference, values are copied to same memory location and hence changes made to formal parameters data are reflected back to caller, since values are copied to same memory locations.

∴ In call by reference, Formal parameters are preceeded by & operator (address) which makes it a Reference parameter. Reference parameters directly accesses the actual arguments in the function call.



DOWNLOADED FROM KTUASSIST.IN

Eg:-

```
#include <stdio.h>
int main()
{
int x=3;
func1(x);
printf("%d", x);
func2(x);
printf("%d", x);
}
void func1(int x)
{
x=20;
}
void func2(int x)
{
x=30;
}
```

Call by Value

3	30
2502	2504

Call by Reference

30
2502

In call by value, Both of the names of formal parameters and actual parameters are independent.

DOWNLOADED FROM KTUASSIST.IN

Eg:. Swapping values of 2 variables

```
#include <stdio.h>
void swap (int, int);
void swap1 (int &, int &);

int main()
{
int a=3, b=5;
printf("Original values\n");
printf ("%d \t %d", a, b);
printf(" Call by Value\n");
swap (a, b);
printf("%d \t %d", a, b);
printf("\n Call by Reference\n");
swap1 (a, b);
printf ("%d\t %d", a, b);

}
```

```
void swap(int x, int y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

```
void swap1(int &x, int &y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

In call by Reference, both the names are aliases for the same object. i.e Changes made through one is visible through the other. Thus, Actual parameters can be modified through call by reference parameter passing technique

DOWNLOADED FROM KTUASSIST.IN

9n) Write a C program to find sum of the series $x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \dots$

```
#include <stdio.h>
int power (int x, int y);
int fact (int N);
int main()
{
float sum;
int term1, term2, x, count;
sum=0.0;
printf(" Enter the value of x and N");
scanf("%d %d", &x, &N);
count = 3;
while (count <= N)
 {
   term1 = power(x, count);
   term2 = fact(count);
   sum = sum + (float) term1 / term2;
   count = count + 2;
 }
```

DOWNLOADED FROM KTUASSIST.IN

```
printf("Sum of series = %f", sum);
}
int power(int x, int y)
{
int i;
int pow=1;
for(i=1; i<=y; i++)
{
pow=pow * x;
return pow;
}
}
int fact (int N)
{
int i, fact=1;
for (i=1; i<=N; i++)
{
fact= fact * i;
}
return fact;
}
```

DOWNLOADED FROM KTUASSIST.IN

Recursion

Recursive functions are those functions that calls to itself many no: of times until terminating condition is satisfied to stop the looping action, otherwise it may result in an infinite loop.

eg:- $n! = n(n-1)(n-2)\ldots$

Factorial of a number

```
#include <stdio.h>
int fact(int);
int main()
{
int x;
printf("Enter the number");
scanf("%d",&x);
```

DOWNLOADED FROM KTUASSIST.IN

```
printf("Factorial = %d", fact(x));
}
int fact(int n)
{
 int f;
 if ((n== 0) || (n == 1))  // Terminating
                          // condition
     return 1;
 else
   f = n*fact(n-1);   // Recursive
   return f;          // function
                      // call.
}
```

O/p

Enter the number 3

Factorial = 6

DOWNLOADED FROM KTUASSIST.IN

Arrays as function parameters

Not only variables of type int, char, float and double can be passed as function arguments, but array parameters as well. When an array is passed as argument to function, only array name is passed. This array name represents the memory address of the first array element in an integer array.

DOWNLOADED FROM KTUASSIST.IN

Eg:-

```
#include<stdio.h>
# [illegible]

void disp (int marks[5]);

int main()
{
int m[]= {89, 90, 75, 70, 67};
disp(m);
}
void disp (int marks[5])
{
for(int i=0; i<5; i++)
{
printf ("Student %d", i+1);
printf ("scored %d marks", marks[i]);
}
}
```

According to eg, actual parameter m is copied to [illegible] formal parameter, int marks[5] in the function definition. This saves memory space and time.

DOWNLOADED FROM KTUASSIST.IN

Qn) Write a program to sort integer and character array using functions?

```
#include<stdio.h>
void sort1(int[] , int);
void sort2(char[], int);
    int main()
    {
    int a[10];
     char b[10];
    int i,n;
printf(" Enter the limit");
scanf("%d", &n);
printf("Enter integer array");
for(i=0; i<n; i++)
 scanf("%d", &a[i]);
printf ("Enter character array");
for (i=0 ; i<n; i++)
scanf ("%d", &b[i]);
sort1(a, n);
sort2(b, n);
}
```

DOWNLOADED FROM KTUASSIST.IN

```
void sort1 (int a[], int n)
{
  int i,j,temp;
  for (i=0; i<n; i++)
  {
  for (j=i+1; j<n; j++)
   {
    if(a[i] > a[j])
     {
      temp=a[i];
      a[i]=a[j];
      a[j]=temp;
     }
  }
}
  printf("Sorted List");
  for(i=0; i<n; i++)
   {
    printf("%d \t",a[i]);
   }
  }
```

```
void sort2 (char b[],
                  int n)
  {
   int i,j;
   char temp;
  for (i=0; i<n ; i++)
   {
  for (j=i+1; j<n; j++)
  {
  if (b[i] >b[j])
    {
    temp = b[i];
    b[i] = b[j];
     b[j]=temp;
    }
  }
}
   printf("Sorted List");
   for(i=0; i<n; i++)
   {
  printf("%c\t", b[i]);
    }
}
```

DOWNLOADED FROM KTUASSIST.IN

5 advantages of using functions are :-

- Functions avoid repetition of code.
- Functions increase program readability.
- Using functions, we can divide complex problem into simpler ones.
- Functions reduce chances of error.
- Program modification is made more easier using functions.

DOWNLOADED FROM KTUASSIST.IN

Introduction to Structures

All are aware that a variable stores a single value of a datatype, while Arrays can store many values of similar datatype. In real life, we need to store different data types together, to maintain employee information compresing of name, age, qualification and salary. For tackling such mixed data types in a variable, we use structures.

A structure is a collection of one or more variables of different data types, grouped together under a single name.

DOWNLOADED FROM KTUASSIST.IN

Features:-

→ It is possible to copy the contents of all structure elements of different datatypes to another structure variable of same data type using assignment operator (=).

It is because structure elements are stored in successive memory locations.

→ Nesting of structures is possible. i.e one can create a structure within a structure. Using this feature, one can handle complex data types.

DOWNLOADED FROM KTUASSIST.IN

→ It is also possible to pass structure elements to a function, either by call-by-value or call-by-address.

→ It is also possible to create a structure pointer pointing to structure elements using member access operator (→).

→ Structure declaration starts with the struct keyword. It defines the structure, but it does not allocate memory. Memory allocation takes place only when variables are declared.

Dot operator is used when simple object is declared and arrow operator (→) is declared, when object is a pointer to a structure.

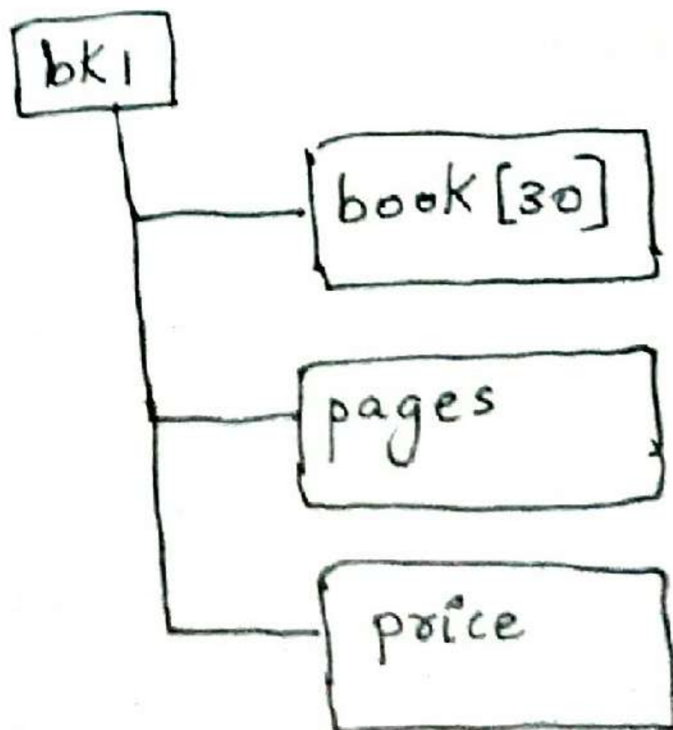
DOWNLOADED FROM KTUASSIST.IN

Eg:-

```
struct book1            //Structures
{                       //hold information
char book[30];
                        // comprising of
int pages;
                        //different
float price;
                        // data types.
}bk1;
```

A structure bk1 of type book1 is created.

Block diagram of structure.



DOWNLOADED FROM KTUASSIST.IN

Syntax

```
struct tagname
  {
    member1;   // individual
    member2;
    .....      // member declarations.
    member n;
  };
```

```
struct tag-name variable-name;
```

Structure variables of type tag are created using struct datatype. No initialization is permitted inside structure.

```
struct student
{
  char name[30];
  int rollno;
  float tot_marks;
};
struct student s;
```

s is a structure variable holding the details of structure student.

DOWNLOADED FROM KTUASSIST.IN

Structure members are accessed using structure variable as follows:-

structure variable.structure member name = value;

Eg:- s.name = "HARI"

s.rollno = 33

s.tot_marks = 92.6;

Eg:-
```
#include<stdio.h>
struct item
{
int codeno;
 float price;
 int qty;
};
int main()
{
struct item a,*b;
```

DOWNLOADED FROM KTUASSIST.IN

```
a.codeno = 123;
a.price = 150.75;
a.qty = 150;
printf("\n Code no: %d",a.codeno);
printf("\n Price:%d",a.price);
printf("\n Quantity:%d ",a.qty);
b->codeno = 124;
b->price = 200.75;
b->qty = 75;
printf( "with pointer to structure");
printf ("Code no: %d", b->codeno);
printf ("\n Price: %f ", b->price);
printf("\n Quantity: %d", b->qty);
}
```

O/P

```
Codeno: 123
Price : 150.75
Quantity:150

with pointer to structure
Codeno : 124
Price : 200.75
Quantity : 75
```

DOWNLOADED FROM KTUASSIST.IN

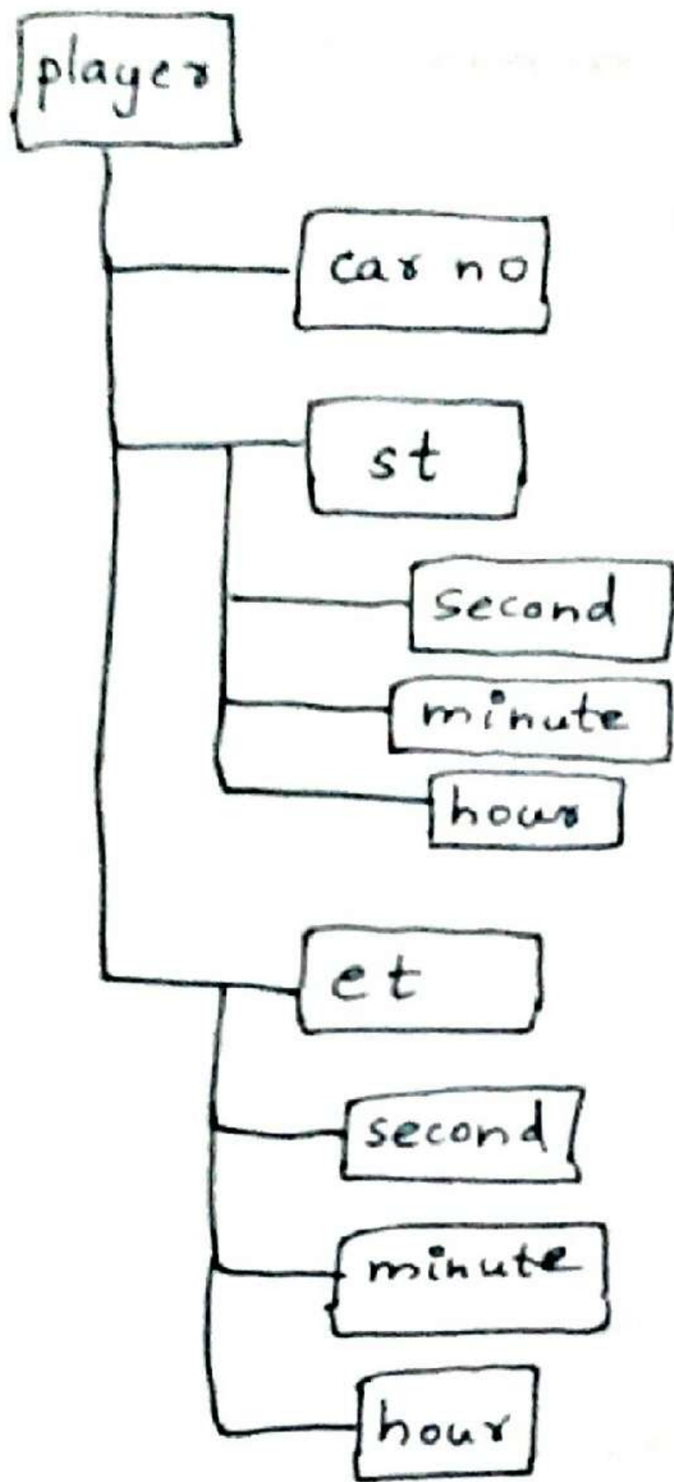
Structure within Structure (Nested Structure)

A structure variable can include another structure variable as its member.

Eg:-
```
struct time
{
 int second;
 int minute;
 int hour;
};

struct t                    //structure t
{
int carno;                  // contains
struct time st;             // another
struct time et;             //structure
};                          // time as one
                            // of its
struct t  player;           // members
```

DOWNLOADED FROM KTUASSIST.IN



Memory allocation of a nested structure

DOWNLOADED FROM KTUASSIST.IN

Qn) Write a program to enter full name and date of birth of a person? Use nested structure?

```
# include <stdio.h>
struct name
{
char first_nm [10];
char last_nm[10];
};
Struct b_date
{
int day;
int mon;
int year;
};
struct data
{
 Struct name nm;
 struct b_date bt;
};
```

DOWNLOADED FROM KTUASSIST.IN

```
int main()
{
 struct data s;
printf ("Enter first name & last name");
 scanf ("%s %s", s.nm.first_nm,
          s.nm.last_nm);
  printf("Enter birth date");
  scanf("%d %d %d",&s.bt.day,
  &s.bt.mon,&s.bt.year);
  printf("Student details:: \n");
  printf("Name\n");
printf("%s", s.nm.first_nm, "%s", s.nm.last_nm);
  printf("Birthdate\n");
  printf("%d / %d / %d", s.bt.day,
          s.bt.mon .s.bt.year);
  }
```

DOWNLOADED FROM KTUASSIST.IN

O/P

Enter first & lastname

Roy Jose

Enter birth date

12 10 1996

Student details.

Name:

Roy Jose

Birth date:

12/10/1996

Array of Structures

In array of structures, every array element is of structure data type. All array elements are stored in adjacent memory

DOWNLOADED FROM KTUASSIST.IN

allocations. Array of structures is a collection of array elements of similar data types, which contains dissimilar data types.

Eg:- Rank list of Students

```
#include <stdio.h>
#include <string.h>
struct student
{
char name[25];
float m1,m2,m3;
char grade[20];
};
int main()
{
struct student s[5];
struct student s1;
float temp,tot[10];
int i,n;
```

DOWNLOADED FROM KTUASSIST.IN

```
printf("Enter the no: of students");
scanf("%d", &n);
for (i=0; i<n; i++)
{
printf(" Enter the name of student %d", i+1);
printf(" Enter the marks in 3 subjects");
scanf("%s", s[i].name);
scanf("%d %d %d", s[i].m1, s[i].m2,
                  s[i].m3);
tot[i]= s[i].m1 + s[i].m2 + s[i].m3;
if (tot[i] > 250)
    strcpy (s[i].grade, "Distinction");
else if (tot[i] >200)
    strcpy (s[i].grade, "First Class");
else if (tot[i]>120)
    strcpy (s[i].grade, "Failed");
}
```

DOWNLOADED FROM KTUASSIST.IN

// Preparation of Ranklist

```
for (i=0; i<n; i++)
{
for(int j=i+1; j<n; j++)
{
if(tot[i] < tot[j])
{
tem = tot[j];
tot[j] = tot[i];
tot[i] = tem;
s1 = s[j];
s[j] = s[i];
s[i] = s1;
}
}
}
printf("Name \t Mark1 \t Mark2 \t Mark3 \t
        Total \t Grade \t \n");

printf("- - - - - - - - - - - - - - - - - \n\n");

for(i=0; i<n; i++)
{
printf("%s \t %f \t %f \t %f \t %f \t %s \t",
s[i].name, s[i].m1, s[i].m2, s[i].m3, tot[i],
        s[i].grade);
} }
```

DOWNLOADED FROM KTUASSIST.IN

O/P

Enter the no: of students 2

Enter the name of student1 Minnu

Enter the marks in 3 subjects 99 87 82

Enter the name of student2 Rose

Enter the marks in 3 subjects 30 51 44

Name	Mark1	Mark2	Mark3	Total	Grade
Minnu	99	87	82	268	Distinction
Rose	30	51	44	135	Second class.

User defined data types (typedef)

It allows users to define new data types equivalent to existing data types.

A new datatype is defined as,

```
typedef type new-type;
```

DOWNLOADED FROM KTUASSIST.IN

where type refers to existing datatype and new-type refers to new user-defined data type.

Eg:-
```
typedef struct
{
    member1;
    member2;
    . . .
    . . .
    member m;
} new-type;
```

In the above eg, new type is a user-defined structure type.

Eg:-
```
typedef struct
{
    int accno;
    char acc-type;
    char name[80];
    float bal;
} record;
record oldcustomer, newcustomer;
```

DOWNLOADED FROM KTUASSIST.IN

Union

Members of a union share same storage area. Union permits several data items to be stored in the same portion of computers memory. Unions are used to conserve memory. Unions are created using union Keyword.

```
union tag
{
  member 1;
  member 2;
   ....
  member n;
};
```

DOWNLOADED FROM KTUASSIST.IN

union tag var-name1, ... var-name n;

Eg:- union id {

```
char color[12];
int size;
} shirt, blouse;
```

Here, we have declared 2 union variables shirt and blouse of type id.

Each individual union member can be accessed using dot operator (.) or member access operator (->).

DOWNLOADED FROM KTUASSIST.IN

Eg:-

```
#include <stdio.h>

int main ()
{
 union id {
      char color;
      int size;
   };
  struct {
   char manufacterer [20];
   float cost;
     union id description;
   } shirt, blouse

printf ("%d\n", size of (union id));
  shirt. description. color = 'w';
 printf ("%c %d \n", shirt.description.color,
        shirt.description.size);
```

DOWNLOADED FROM KTUASSIST.IN

```
shirt.description.size = 12;
printf ("%c %d \n", shirt.description.color,
        shirt.description.size);
}
```

Output

2 → 2 bytes memory allocation (largest members size) int - 2 bytes

W -24713

@ 12

Only one member of union can be assigned a value at any 1 time. Most compilers accept only an initial value for 1 union member.

DOWNLOADED FROM KTUASSIST.IN

Storage Classes in C

Storage Classes controls two different properties of a variable : lifetime and scope.

Two types of scope :- (a) local
(b) global

Local scope is limited to the function, where it is defined. The life of a local variable ends when the function exits.

Global scope covers the entire program. Global variables are defined outside all functions just below header file declaration. Its lifetime ends, only when the program ends.

DOWNLOADED FROM KTUASSIST.IN

```c
#include <stdio.h>
int c = 12; //Global variable initialization
void test();
int main()
{
int d=5; //Local variable initialization
++d;
++c;
printf("d = %d", d);
printf("\n c = %d",c);
test();
}
void test()
{
++c;
printf("\n c = %d", c);
}
```

Output

d = 6
c = 13
c = 14

DOWNLOADED FROM KTUASSIST.IN

Storage Classes

The 4 storage classes in C are auto, static, register and extern.

(a) auto -

Automatic variables are declared using keyword auto. However, if a variable is declared without any keyword inside a function, it is automatic by default. auto storage class applies to local variables only. This automatic variable is visible only within the function where it is declared and its lifetime is the same as the lifetime of the function as well. Once the execution

DOWNLOADED FROM KTUASSIST.IN

of function is finished, variable is destroyed.

Syntax of automatic storage class declaration -

auto data-type var-name = value;

Eg:- auto int x = 2;

(b) Static -

Static local variables exists only inside a function, where it is declared and defined. Its lifetime starts when the function is called and ends only after the program execution has finished. When a function is called, static variable defined

DOWNLOADED FROM KTUASSIST.IN

inside the function, retains its previous value and operates on it. The default initial value of a static variable is 0.

Syntax of static storage class <u>declaration</u> -

static datatype variablename=value;

static int x = 100;

Eg:-

```
#include <stdio.h>
int main()
{
test();
test();
}

void test()
{
static int var=0;
++var;
printf("%d", var);
}
```

<u>Output</u>

1
2

DOWNLOADED FROM KTUASSIST.IN

(c) regester -

register storage class assigns a variables storage in the CPU registers rather than the primary memory. register variables has its lifetime and visibility same as automatic variables. The purpose of creating register variable is to increase the access speed and to make the program run faster. If there is no space available in the register, these variables occupy space in the main memory and

DOWNLOADED FROM KTUASSIST.IN

act similar to automatic storage class variables. Only those variables which require faster access is made as register.

Syntax of Register storage class declaration -

register datatype variable-name = value;

Eg:- register int d;

(d) extern -

External storage class assigns the variable with a reference to the global variable declared outside the given program.

DOWNLOADED FROM KTUASSIST.IN

extern Keyword is used to declare external variables. These variables are visible throughout the program and its lifetime is same as the lifetime of the program where it is declared.

Syntax of external storage class declaration -

```
extern datatype var-name = value;
```

Eg :- extern int test;

Storage Class	Keyword	Life time	visibility	Initial value
Automatic	auto	Function block	Local	Garbage value
External	extern	Whole program	Global	Zero
Static	static	Whole program	Local	zero
Register	register	Function block	Local	Garbage value

DOWNLOADED FROM KTUASSIST.IN

Eg:- File sub.c

```
int test = 100;
void multiply (int n)
{
test = test * n;
}
```

File : main.c

```
#include <stdio.h>
#include "sub.c"
extern int test;
int main()
{
printf("%d", test);
multiply(5);
printf("\n%d", test);
}
```

Output

```
100
500
```

Variable test is declared as external in main.c. This variable can be accessed from both programs. The program performs multiplication and changes global variable test to 500.

DOWNLOADED FROM KTUASSIST.IN