Nadar Saraswathi College of Engineering & Technology

NSCET E-LEARNING PRESENTATION

LISTEN … LEARN… LEAD…

Department of Electronics and Communication Engineering

II YEAR / III SEMESTER

EC8393 – FUNDAMENTALS OF DATA STRUCTURES IN C

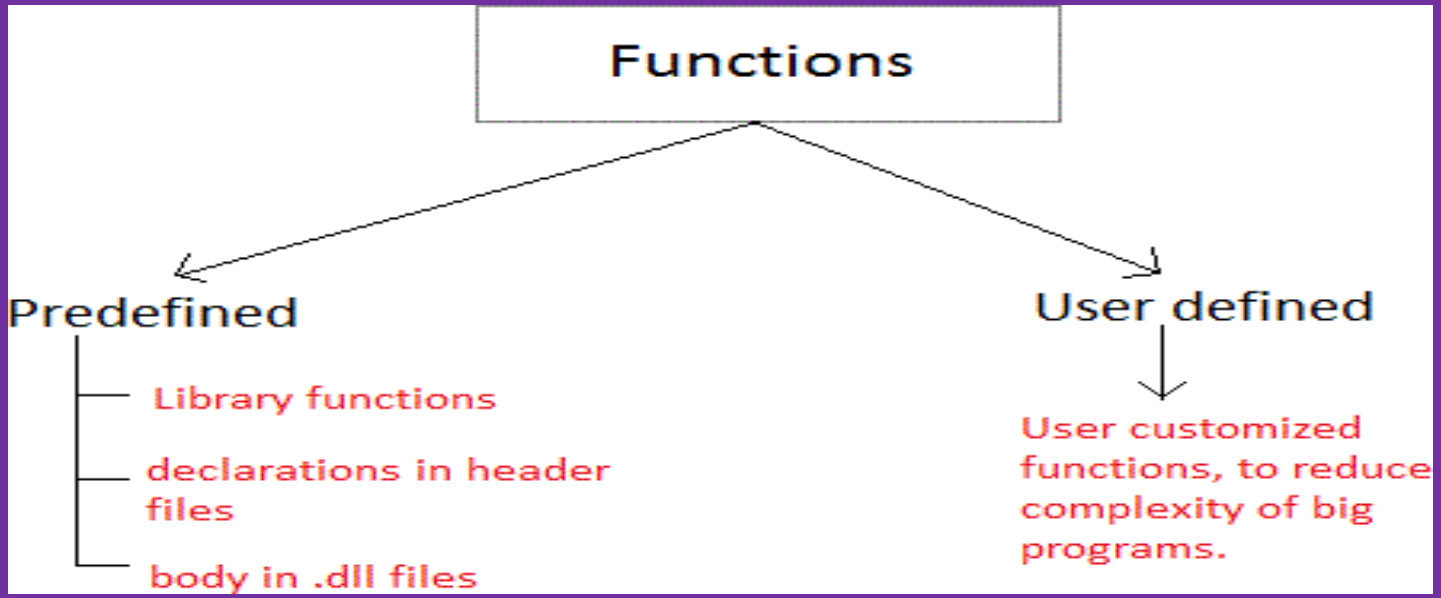
Mr.P.G.Siva Sharma Karthick M.E.,MBA.,
Assistant Professor
Nadar Saraswathi College of Engineering & Technology,
Vadapudupatti, Annanji (po), Theni – 625531.

UNIT-2

FUNCTIONS, POINTERS, STRUCTURES AND UNIONS

FUNCTIONS

- A **function** is a block of code that performs a particular task.
- C functions can be classified into two categories,
  - **Library functions**
  - **User-defined functions**


Functions
Predefined
Library functions
declarations in header files
body in .dll files
User defined
User customized functions, to reduce complexity of big programs.

Benefits of Using Functions

- It provides modularity.
- It makes your code reusable.
- Debugging and editing becomes easier if you use functions.
- It makes the program more readable and easy to understand.

Function Declaration/ Function Prototyping

returntype functionName(type1 parameter1, type2 parameter2,...);

- Function declaration informs the compiler about the function name, parameters is accept, and its return type.
- The actual body of the function can be defined separately. It's also called as. Function declaration consists of 4 parts.
  - returntype
  - function name
  - parameter list
  - terminating semicolon

returntype

- Return type specifies the type of value(int, float, char, double) that function is expected to return to the program which called the function.

functionName

- Function name is an identifier and it specifies the name of the function.

parameter list

- The parameter list declares the type and number of arguments that the function expects when it is called.
- the parameters in the parameter list receives the argument values when the function is called. They are often referred as **formal parameters**.

Function definition

```
returntype functionName(type1 parameter1,
   type2 parameter2,...)
 {
// function body goes here
 }
```

Function call

functionName(argument1, argument2,...);

- When a function is called, control of the program gets transferred to the function.

Types of function implementation

1.Passing nothing and returning nothing

2.Passing the parameters and returning nothing

3.Passing parameters and returning something

Pass by value: #include<stdio.h>
#include<conio.h> void main()
{
int a=10,b=20; void sum(int , int); clrscr();
sum(a,b);
getch();
}
void sum(int x,int y)
{
int c; c=x+ y;
printf("c value is%d",c);
}
OUTPUT:
C value is 30

Pass by reference:

```
#include<stdio.h>
#include<conio.h>  void main()
{
int a=10,b=20;
void sum(int *, int *);
clrscr();
sum(&a,&b);
getch();
}
void sum(int *x,int *y)
{
int c;  c=*x+ *y;
printf("c value is%d",c);
}
```

OUTPUT:

C value is 30

```
#include<stdio.h>
int f(int); // function declaration  int main()
{
  int n, i = 0, c;  scanf("%d", &n);
  printf("Fibonacci series terms are:\n");
  for (c = 1; c <= n; c++)
  {
    printf("%d\n", f(i)); // function call  i++;
  }
  return 0;
}
int f(int n)         // function definition
{
  if (n == 0 || n == 1)  return n;
  else
    return (f(n-1) + f(n-2));
}
```

Structure

- Structure is a **user-defined datatype** in C language which allows us to **combine data of different types together.**
- **structure** is another user defined data type available in C that allows to combine data items of different kinds.
- **Collection of different datatypes.**

Declaration - syntax

```
struct  name
 {
 member definition;
member definition;
} one or more structure variables;
```

example

- Define student structure with members of roll no, name and marks

Method1:

```
struct student
 {
char name[50];
int rollno;
 float marks
 }s1;
```

Method2:

```
struct student
 {
char name[50];
int rollno;
 float marks
 };
struct student s1;
```

EXAMPLE:

```
#include<stdio.h> #include<conio.h> struct student
{
int rollno; char name[5];
float m1,m2,m3,m4,m5,total,avg;
}s1;
void main()
{
clrscr();
printf("enter the student name"); scanf("%s",&s1.name); printf("enter roll no");
scanf("%d",&s1.rollno); printf("enter marks");
scanf("%f%f%f%f%f",&s1.m1,&s1.m2,&s1.m3,&s1.m4,&s1.m5); s1.total= s1.m1+s1.m2+s1.m3+s1.m4+s1.m5;
s1.avg=s1.total/5; printf("name is%s",s1.name); printf("roll no is%d",s1.rollno);
printf("marks are %f%f%f%f%f","s1.m1,s1.m2,s1.m3,s1.m4,s1.m5);
printf("total is %f",s1.total); printf("average is%f",s1.avg); getch();
}
```

OUTPUT:

enter the student name
manju
enter roll no
56
enter marks
56 67 78 98 69
total is
368
average is
73.6

Define a structure to store details of 10 bank customers with customer with name, account number, balance and city. Write a C program to store the details of customers in the bank, access and print the customer details for specified account number.

```
#include<stdio.h>          #include<conio.h>
struct customer
{
char c_name[20];  int acc_no;
int bal;
char city[20];
}c[10];
```

```
void main()
{
int i,acc;  clrscr();  for(i=0;i<10;i++)
{
Printf(“enter the cust %d details”,i+1);  printf("enter the student
name");  scanf("%s",c[i].c_name);
printf("enter acc no");  scanf("%d",&c[i].acc_no);
printf("enter balance");  printf("%d”,&c[i].bal);
printf("enter the citty name");  scanf("%s",c[i].city);
}
```

```
printf("enter the acc.no to display the details”);
scanf(“%d”,&acc);  for(i=0;i<10;i++)
{
     if(c[i].acc_no == acc)
     {
printf(“cust name is %s”,c[i].c_name);  printf("balance is
%d”,c[i].bal);
                              printf(“city is %s”,c[i].city);
               }
     }
}
getch();
}
```

Define a structure called book name, author name, and price. Write a C program to read the details of book name, author name and price of 200 books in a library and display the total costs of the books and the book details whose price is above Rs.500

```
int price;
char b_name[50];
char a_name[50];
}b[200];
```

```
#include<stdio.h>
#include<conio.h>
struct book
        {
```

```
void main()
{
int sum=0;  clrscr();  for(i=0;i<200;i++)
{
Printf(“enter the book %d details”,i+1);  printf("enter the book name");
scanf("%s",b[i].b_name);
printf("enter the author name");  scanf("%s",b[i].a_name);

printf("enter amt");
scanf("%d",&b[i].price);
}
for(i=0;i<200;i++)
{
Sum=sum+b[i].price;
}
```

```
Printf("total price of books is %d",sum);  Printf("books
whose price is above 500);  for(i=0;i<200;i++)
{
    if(b[i].price>500)
            printf("%s",b[i].b_name);
}
getch();
}
```

Structure within structure

- Nested structures
- Nested structures means, that one structure has another structure as member variable.

```
struct Student {  char[30]
name;  int age;
/* here Address is a structure
*/
struct Address
 {
 char[50] locality;  char[50]
 city;
 int pincode;
}addr;
};
```

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

struct student_college_detail
{
    int college_id;
    char college_name[50];
};

struct student_detail
{
    int id;
    char name[20];  float
    percentage;
    // structure within structure
    struct student_college_detail clg_data;
}stu_data;
```

```
int main()
{
    struct student_detail stu_data = {1, "Raju", 90.5, 71145,
                                      "Anna University"};  printf("
    Id is: %d \n", stu_data.id);  printf(" Name is: %s \n",
    stu_data.name);
    printf(" Percentage is: %f \n\n", stu_data.percentage);

    printf(" College Id is: %d \n",
                    stu_data.clg_data.college_id);  printf("
    College Name is: %s \n",
                    stu_data.clg_data.college_name);
    return 0;
}
```

OUTPUT:

```
Id is: 1
Name is: Raju  Percentage is:
90.500000  College Id is: 71145
College Name is: Anna University
```

recursion

- Programming technique
- **Function calls itself repeatedly** for some input

Properties:

- Must be at least one condition ❼ base case property
- Invoking of each recursive call must reduce to some manipulation and must go closer to base case condition.

Example2:

```
#include<stdio.h>  #include<conio.h>
long factorial(int); // function declaration  int main()
{
  int n;  long f;
  printf("Enter an integer to find its  factorial\n");
  scanf("%d", &n);  if (n < 0)
    printf("Factorial of negative integers isn't defined.\n");  else
  {
    f = factorial(n);  printf("%d! = %ld\n", n, f);
  }

  getch();
}

long factorial(int n)
{
  if (n == 0)  return 1;  else
    return(n * factorial(n-1));
}
```

pointer

- Pointer is a variable that represents the memory location of some other variable
- Hold the memory location and not the actual value

*** Pointer variable**

•Example : int *p;

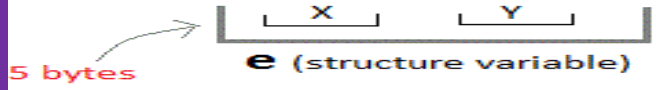
```
#include <stdio.h>  int
main()
{     int x, y, *a, *b, temp;
      printf("Enter the value of x and y\n");  scanf("%d%d",
      &x, &y);
   printf("Before Swapping\nx = %d\ny = %d\n", x, y);  a = &x;       b = &y;temp
= *b;              *b = *a;
  *a = temp;
printf("After Swapping\nx = %d\ny = %d\n", x, y);  return 0;
}
```

UNION

- **Unions** are conceptually similar to **structures**. The syntax to declare/define a union is also similar to that of a structure. The only differences is in terms of storage.
- In **structure** each member has its own storage location, whereas all members of **union** uses a single shared memory location which is equal to the size of its largest data member.

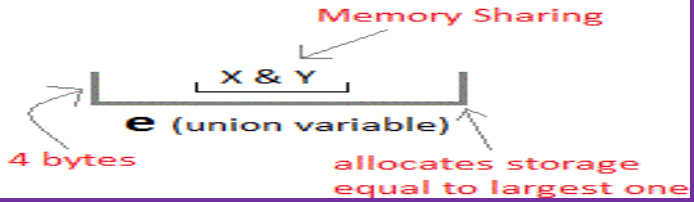
Structure

```
struct Emp
{
 char X ;      // size 1 byte
 float Y ;     // size 4 byte
} e ;
```



Unions

```
union Emp
{
 char X ;
 float Y ;
} e ;
```



Difference Between Structure & 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.

```
union item
{
 int m;
float x;
 char c;
 }It1;
```

```
#include <stdio.h>
union item
{ int a;
 float b;
 char ch;
 };
 int main( )
 { union item it;  it.a
 = 12;
it.b = 20.2;
 it.ch = 'z';
 printf("%d\n", it.a); printf("%f\n", it.b);
 printf("%c\n", it.ch);
return 0;
 }
```

Output:

-26426

20.1999

z

Storage classes

- In C language, each variable has a storage class which decides scope and lifetime of that variable.
- The scope of a declaration is the part of the program for which the declaration is in effect.
- The lifetime of a variable or object is the time period in which the variable/object has valid memory.
- 4 types of storage classes
  - AUTO
  - REGISTER
  - STATIC
  - EXTERN

Auto:

- A variable declared inside a function without any storage class specification, is by default an **automatic variable**.
- Automatic variables are local to a function.
- They are created when a function is called and are destroyed **automatically** when the function exits.

Ex:

int detail; or auto int detail; //Both are same

Register:

- **Register** variable inform the compiler to store the local variables in register instead of memory.
- **Register** variable has faster access than normal variable.
- We cannot access the address of such variables since these do not have a memory location

Syntax:

register int number;

Static:

- Instead of creating and destroying a variable every time when it comes into and goes out of scope, **static** is initialized only once and remains into existence till the end of program
- They are assigned **0 (zero)** as default value by the compiler.

```
void test(); //Function declaration
main()
{
    test();
    test();
    test();
}
 void test() {
static int a = 0;  a
= a+1;
printf("%d\t",a);
                    }
```

output : 1 2 3

Extern:

- If a variable is declared with global scope in one file but referenced in another,then extern keyword is used to inform the compiler of the variable's existence.
- An extern declaration does not create any storage
- file1.c

```
#include<stdio.h>
int a=10;
Void fun()
{…}
file2.c
    #include "file1.c"
    main()
    { extern int a;
            fun();
     }
```

Preprocessor Directives

- Pre-processing is a step applied on the source file before presenting it to compilation process
- Pre-processor is a program which performs preprocessing.
- Pre-processor directive is written at the beginning of the program with # preceded to it.
- List of preprocessor directives
- Macro
- File Inclusion
- Conditional compilation
- Other directives

- Macro
  - It defines symbolic constants with its values
  - It can also have arguments,same as functions
  - Syntax: #define
  - Ex-1:
    - #define PI 3.14
  - Ex-2:
- **#define AREA(x) ((PI)*(x)*(x))**
- **area = AREA(4);**
- is expanded to
- **area =(3.14*(4)*(4));**

- File Inclusion
  - The source code of the file “file_name” is included in the program
  - Syntax: #include<file_name>
    - » Searches standard library for file
    - » Use for standard library files

#include “file_name”

>> Use for user-defined files

- Ex-1: #include<stdio.h>
- Ex-2: #include “sample.c”

- Conditional compilation
  - Set of commands are included or excluded in source program before compilation with respect to the condition.
  - Syntax:

- #ifdef - If this macro is defined
- #ifndef - If this macro is not defined
- #if - Test if a compile time condition is true
- #else - The alternative for #if
- #elif
- #endif - End preprocessor conditional

Ex-1:

- #ifndef MESSAGE
- #define MESSAGE "You wish!"
- #endif

EX-2:

```
#define NO 7
#if NO==0
    #define STACK 0
#elif NO==1
    #define STACK 100
#else
    #define STACK 200
#endif
```

- Other directives
  - Syntax: #undef
    - It is used to undefine a defined macro variable
  - Syntax: #pragma
    - The pragma directive is used to access compiler-specific features.
    - Ex:
- #pragma loop_opt(off)