UNIT – III
Introduction To C

Unit 3 Contents

 Introductions to C: Fundamentals of C-Programming - Character Set, Identifiers and keywords, Data types, Constants, Variables, Operators, Expression, statements, Library Functions, Pre-processor directives. Data Input and Output, Control Structures- Decision making, Control Structures-Iterative, break and continue statements, Structure of C program, Coding conventions. Array-single, multidimensional arrays, String in C –standard string functions in string.h. **Functions in C, recursion, Different parameter passing methods, Lifetime of variables, Scope rules: Static and Dynamic scope, user defined string functions, Structure, Pointers, Structure - Array of structure, Union, Pointers, Pointers and arrays, Dynamic allocation and its application, Files: Types of File, File operation, Processing File.**
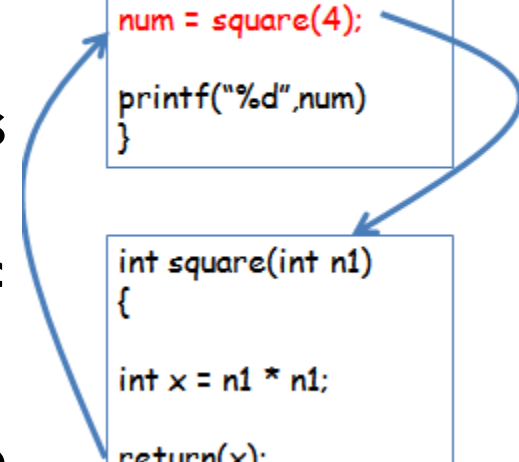
What is a Function

☐ Function is a procedure or a routine that executes a certain task and returns a value

☐ It is a subprogram or a set of instructions written to do a particular task

☐ A function is a 'self-contained' block of statements that perform some task

☐ A large program can be divided into functions and combined into single unit

☐ Every C program is a collection of one or more functions

```
void main()
{
int num;

num = square(4);

printf("%d",num)
}
```

```
int square(int n1)
{

int x = n1 * n1;

return(x);
}
```

Types of Functions

- ☐ Library Functions
  - ◻ They are in-built functions of 'C' library. These are already defined in header files.
  - ◻ Eg. printf(); Defined in stdio.h

    strlen(); Defined in string.h
- ☐ User Defined Functions
  - ◻ Programmer creates his own functions to perform some task
  - ◻ Eg. add()

Library functions

- ☐ These functions are not required to be written by the user

- ☐ They need not be declared and defined

- ☐ To use library functions, add #include preprocessor directive in the program

- ☐ Example:

  - ◼ To use mathematical functions write

    #include<math.h>

  - ◼ To use string functions write

    #include<string.h>

Prototype of User Defined Function

□ Following are the three main sections which must be used while including function

- ◻ Function declaration

- ◻ Function call

- ◻ Function definition

Function Declaration

□ A function name preceded by its return type and followed by its parameter list is called a '*function declaration*' or '*function prototype*'

□ Similar to variable declaration

□ If there is a need of function, it should be declared first

□ Syntax:

data_type function_name (argument list with data types);

here *data_type* is a data type of return value;

function_name is any user given name

argument list is a list of variables required in function

Function Declaration

- Example:

    int add(int a,int b);

    void display();

- The functions can be declared above the main() or in the main() method

- If function does not return any value then *void* data type is used

- Function returns only one value

Function Call

- ☐ It means calling a function in a program by writing function name and passing arguments

- ☐ Syntax:

  function_name(list of actual argument);

  Here *actual arguments* are the values of variables passed / given to the function definition

Function Call

□ Example:

    add(a,b);

    fact(6);

    display();

□ When compiler encounters function call, the control is transferred to the function

□ The function is then executed line by line

□ A value is returned when a return statement is encountered

Types of Function Call

□ Functions communicate with each other by passing arguments.

□ Arguments can be passed in one of following 2 ways:

   □ **Call By Value**

   □ **Call By Reference**

Call By Value

- When function is called by passing normal values or variables then function call is called as a **Call by Value**

- In C, by **default** all function arguments are passed **By Value**.

- When arguments are passed to called function, values are passed through temporary variables. All manipulations are done on these temporary variables only.

- Therefore called function can not access actual memory location of original variable and so can not change its value.

- Example:

    add(5,10);

    add(x,y);

Call By Reference

□ At the time of function call instead of passing variables or values, reference of variables (address of variables) are passed then it is called as **Call by Reference**

□ Here the called function has access to actual memory location of variable passed as an argument

□ Therefore can change value of arguments of calling routine

□ Here & (ampersand) is used before variable when it is passed as a argument in function
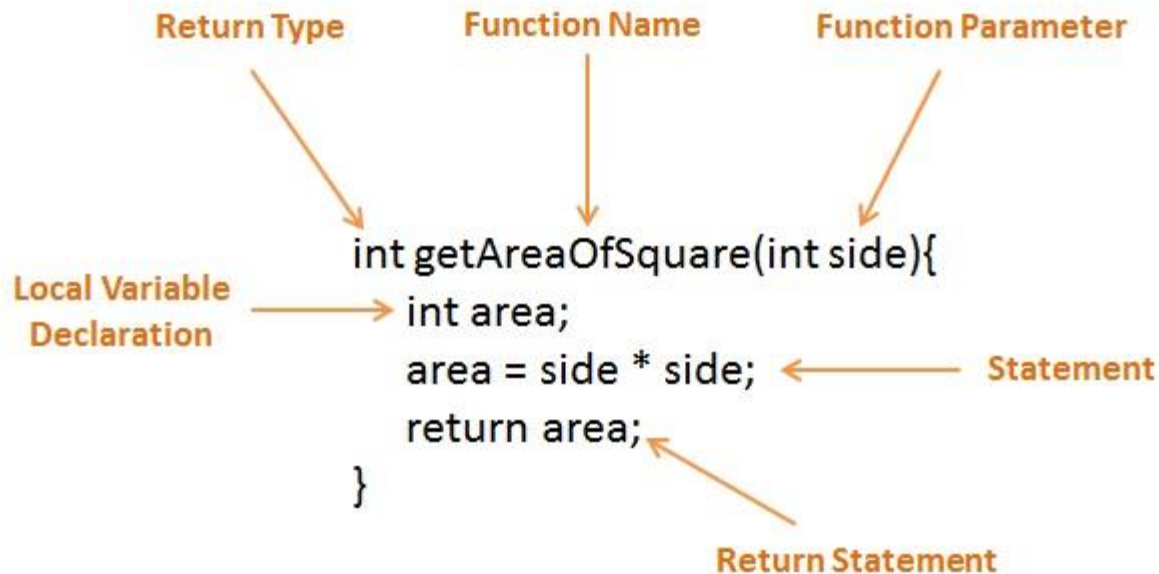
□ Example:

   add(&x,&y);

Function Definition

□ The function's return type, followed by its name, parameter list, and body constitute the '*function definition*'

□ Function name must be same in declaration, call and definition

- ▫ Each function has a *unique name*

- ▫ '*Function body*' refers to the statements that represent the actions that the function performs

Function Definition

Function Definition

The general form of a function definition in C programming language is as follows −

Syntax:

return_type function_name(parameter list) { body of the function }

Function Definition

- **Body of function**: may consist of variable declarations and one or more executable statements

- Result of a function is called as '*return value*'

- The data type of the return value is called the '*return type*'

- If no data type is specified, function is assumed to be void i.e. it returns no result.

- A semicolon is used after function declaration and function call, but not after the function definition.

- Parenthesis are compulsory after the function name, irrespective of whether the function has arguments or not

Function Arguments

- A '*function argument*' is an expression that is used within the parenthesis of a function call.

- **Arguments** are separated by commas. Arguments are enclosed in a parentheses. If no arguments, a pair of empty parentheses must follow the function name.

- Types of Arguments:
    - Actual Arguments
    - Formal Arguments

Function Arguments

- Arguments in calling function are called **Actual Arguments.**

  Called Function is: int add(int x, int y).

- Arguments in called function are called **Formal Arguments**

- Actual arguments are passed in function call and received in formal arguments at function definition

- **Data type** of Actual and Formal arguments should be the **same.**

- **Number and order** of actual arguments should also be the **same** as that in formal arguments.

Program for Addition by using Function

```
#include<stdio.h>
int add(int x , int y);    /* Function
    Declaration / Prototype*/
void main()       /* Calling Function*/
{
     int i;
     printf("Enter two numbers : ");
     scanf("%d%d",&a,&b);
     c=add(a,b);          /*Function
     Call*/
     printf("\n\n\tAddition of %d and
     %d is %d",a,b,c);
}
```

```
int add(int x,int y)  /*Function
 Definition (Called Function)*/
{
        int z;
        z = x + y;
        return(z);
}
```

The 'return' Statement

□ **Syntax:**

return(expression);   *OR*

return(constant);   *OR*

return;

□ **Example:**

return(c=a*b);   OR   return(c);

□ It always returns the value of the expression or variable which is written inside the parenthesis following the return statement.

□ When a function does not return any value, then **void** is used.

□ return statement can return only one value.

The 'return' Statement

☐ It is used to return from a function.

☐ It causes execution to return to the point at which the call to the function was made.

☐ **int add(int x,int y)** can be written in 2 ways:

int add(int x , int y)	int add(int x , int y)
{	{
return(x + y);	int z;
}	z = x + y;
	return(z);
	}

Recursion

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.

- The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

- Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Recursion Example : Fibonacci numbers

```c
//Calculate Fibonacci
  numbers using recursive
  function.
//A very inefficient way,
  but illustrates
  recursion well
int fib(int number)
{
    if (number == 0) return
    0;
    if (number == 1) return
    1;
    return (fib(number-1) +
    fib(number-2));
}

int main(){ // driver function
    int inp_number;
    printf("Please enter an
    integer:");
    scanf("%d",&inp_number);
    printf("The Fibonacci number
    for %d is %d", inp_number ,
    fib(inp_number));
    return 0;
}
```
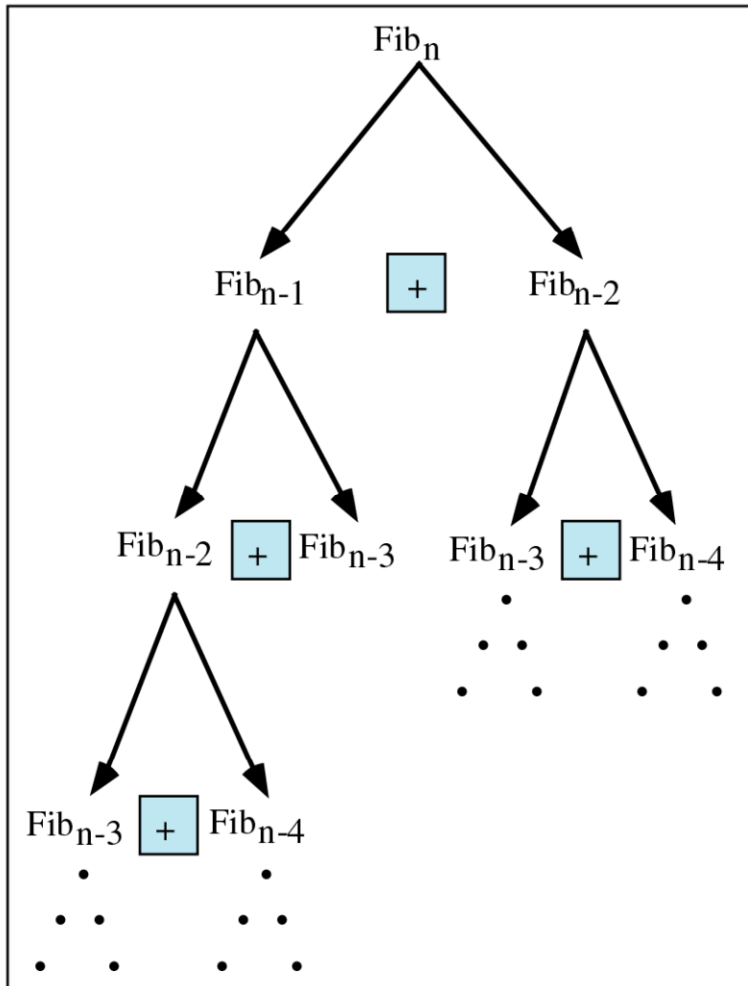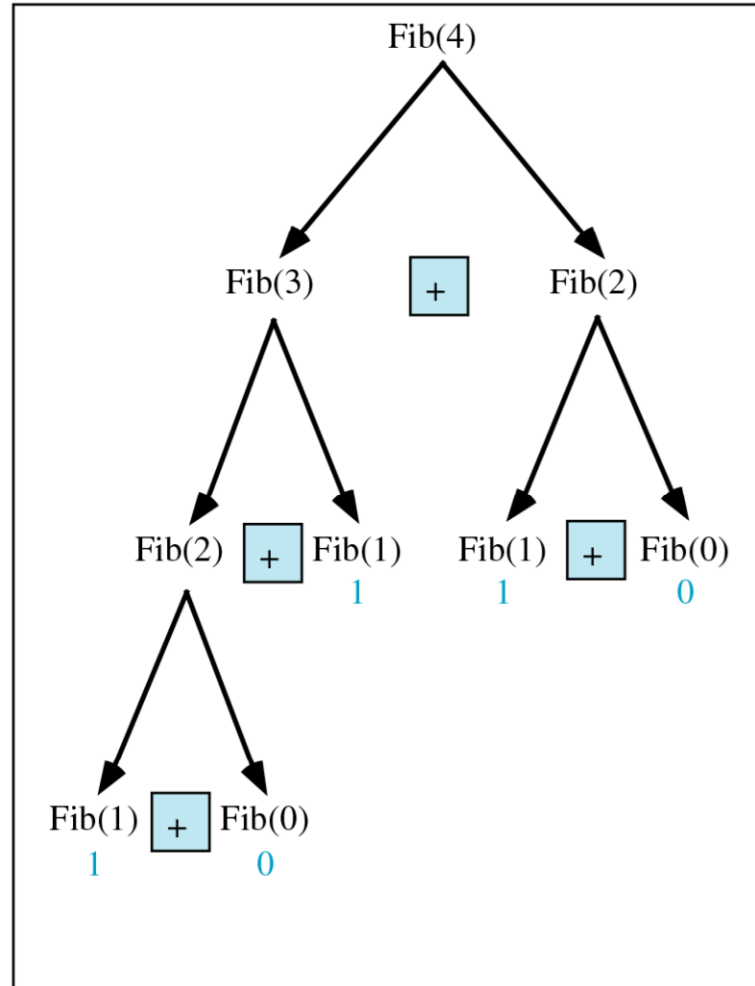
Example contd

(a) Fib(n) (b) Fib(4)

Recursion Example(Factorial)

```c
#include<stdio.h>

int find_factorial(int);

int main() {
    int num, fact; /*Ask user for the
    input and store it in num*/

    printf("\nEnter any integer
    number:");

    scanf("%d",&num);
    /*Calling our user defined
    function*/

    fact =find_factorial(num);
    /*Displaying factorial of input
    number */

    printf("\nFactorial of %d is: %d",num,
    fact);
    return 0;
}
int find_factorial(int n) {
/*Factorial of 0 is 1*/
        if(n==0) return(1);
        /*Function calling itself:
            recursion */
        return(n*find_factorial(n-1));
}
```

Output:
Enter any integer number: 4
Factorial of 4 is: 24

Lifetime of Variables

- Scope is how far a variable is accessible and life is how much time does a variable exists in the memory (life of variable).

- The scope and life of a variable depends on the location where a variable is declared

- According to their declaration, variables are classified into 3 categories
  - Block variables
  - Internal or Local variables
  - External or Global variables.

Block variables in C

Scope of block variables:

☐ Block variables can be accessed within the block in which they are declared, can also be accesses into the inner block which is within the current block but, can't be accessed outside the block.

Life of block variables:

☐ These variables appear as the control enters into the block and disappears as the control go out of the block. Hence these variables can't be accessed outside the block.

Block Variable Example

Output:
x=10
x=10

```c
/* scope of block variables */
#include<stdio.h>
int main(){
 {                    /* outer block */
    int x=10;
    {               /* inner block */
        printf("x=%d",x);
    }
    printf("\nx=%d",x);
 }
 return 0;
}
```

Block Variable Example

```c
/* scope of block variables */
#include<stdio.h>
int main()
{
    if(10<20)
    {
        int x=10;
        printf("x=%d",x);
    }
    printf("\nx=%d",x);  /* can't be accessed */
    return 0;
}
```

Output:
    Error: Undefined symbol "x"
    in function main()

Local Variables

- Variables that are declared inside a function or block are called local variables.

- They can be used only by statements that are inside that function or block of code.

- Local variables are not known to functions outside their own.

Local Variables Example

```c
#include <stdio.h>
int main () {
    /* local variable declaration */
    int a, b; int c;
    /* actual initialization */
    a = 10; b = 20;
    c = a + b;
    printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
    return 0;
}
```

Global Variables

□ Global variables are defined outside a function, usually on top of the program.

□ Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

□ A global variable is available for use throughout your entire program after its declaration.

Global Variables Example

```c
#include <stdio.h>
/* global variable declaration */
int g;
int main () {
    /* local variable declaration */
    int a, b;
    /* actual initialization */
    a = 10; b = 20;
    g = a + b;
    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
    return 0;
}
```

Scope Rules

□ In computer programming, a scope is the context within a computer program in which a variable name or other identifier is valid and can be used, or within which a declaration has effect.

Static Scoping

- Static scoping is also called **lexical scoping.**

- A variable always refers to its top level environment.

- In most of the programming languages including C, C++ and Java, variables are always statically (or lexically) scoped i.e., **binding of a variable can be determined by program text and is independent of the run-time function call stack.**

Static Scoping Example

```c
#include<stdio.h>
int x = 10;
 /* Called by g()*/
int f(){
  return x;
}
/* g() has its own variable*/
/* named as x and calls f()*/
int g(){
  int x = 20;
  return f();
}
```

```c
int main()
{
  printf("%d", g());
  printf("\n");
  return 0;
}
```

Output
 10

Dynamic Scoping

- [ ] With dynamic scope, a global identifier refers to the identifier associated with the most recent environment, and is uncommon in modern languages.

- [ ] In dynamic scoping the compiler first searches the current block and then successively all the calling functions.

Dynamic Scoping Example

/* Since dynamic scoping is very uncommon in the familiar languages, we consider the following pseudo code as our example. It prints 20 in a language that uses dynamic scoping.   */

int x = 10;

/* Called by g()*/

int f(){

   return x;

}

/* g() has its own variable named as x and calls f()*/
int g(){
   int x = 20;
   return f();
}
main(){
  printf(g());
}

Output in a language that uses Dynamic Scoping :

         20

Introduction to Structures

☐ A structure contains a number of data types grouped together.

☐ These data types may or may not be of the same type.

☐ The following example illustrates the use of this data type.

struct book

{

    char name ;

    float price ;

    int pages ;

} ;

struct book b1, b2, b3 ;

This statement defines a new data type called **struct book.** Each variable of this data type will consist of a character variable called **name,** a float variable called **price** and an integer variable called **pages.**

Once the new structure data type has been defined one or more variables can be declared to be of that type.
The variables **b1, b2, b3** can be declared to be of the type **struct book,**

Continue….

We can combine the declaration of the structure type and the structure variables in one statement.

```
struct book
{
        char name ;
        float price ;
        int pages ;
} b1, b2, b3 ;
```

Like primary variables and arrays, structure variables can also be initialized where they are declared.

```
struct book {
        char name[10] ;
        float price ;
        int pages ;
} ;
struct book b1 = { "Basic", 130.00, 550 } ;
struct book b2 = { "Physics", 150.80, 800 } ;
```

How Structure Elements are Stored

Whatever be the elements of a structure, they are always stored in contiguous memory locations. The following program would illustrate this:

```
/* Memory map of structure elements */
main( )
{
struct book
{
    char name ;
    float price ;
    int pages ;
} ;
struct book b1 = { 'B', 130.00, 550 } ;
printf ( "\n Address of name = %u", &b1.name ) ;
printf ( "\n Address of price = %u", &b1.price ) ;
printf ( "\n Address of pages = %u", &b1.pages ) ;
}
```

e.g. The output of the program... Address of name = 65518
Address of price = 65519
Address of pages = 65523

Nested Structures

- Nesting of structures, is also permitted in C language.

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

Example:

struct Student {

char[30] name;

int age;

/* here Address is a structure */

struct Address {

      char[50] locality;

      char[50] city;

      int pincode; }

addr;

};

Array of Structures

If we want to store data of 100 books then we would be required to use 100 different structure variables from b1 to b100, which is definitely not very convenient.

A better approach would be to use an array of structures.

Program shows how to use an array of structures.

```c
main( )
{
   struct book
   {
      char name ;
      float price ;
      int pages ;
   } ;
struct book b[100] ;
int i ;

for ( i = 0 ; i <= 99 ; i++ )
 {
    printf ( "\n Enter name, price and pages " ) ;
    scanf ( "%c %f %d", &b[i].name, &b[i].price,
&b[i].pages ) ;
 }
 for ( i = 0 ; i <= 99 ; i++ )
 printf ( "\n %c %f %d", b[i].name, b[i].price, b[i].pages )
;
}
```

Introduction to Unions

- □ Unions are derived data types, the way structures are.

- □ Both structures and unions are used to group a number of different variables together.

- □ But structure stored a different variables at different spaces in memory.

- □ A union stored a different variables at same space in memory.

Syntax:

```
union demo{
    short int i;
     char ch[2];
};
union demo key;
```

Example of Unions

```c
#include<stdio.h>
int main()
 {
  union a
     {
   short int i;
    char ch[2];
     };
union a key;
key.i = 512;
```

```c
printf("key.i= %d\n", key.i);
printf("key.ch[0]= %d\n",
key.ch[0]);
printf("key.ch[1]= %d\n",
key.ch[1]);
key.ch[0] = 50;
printf("key.i= %d\n", key.i);
printf("key.ch[0]= %d\n",
key.ch[0]);
printf("key.ch[1]= %d\n",
key.ch[1]);
return 0;}
```
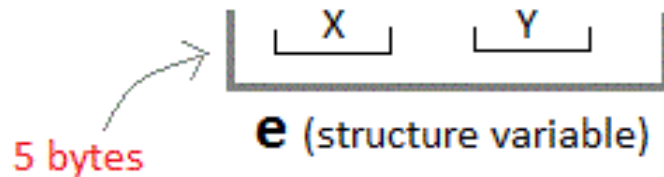
Output:
key.i=512

key.ch[0]=0

key.ch[1]=2

key.i=50

key.ch[0]=50

key.ch[1]=2

Structure

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

X   Y

e (structure variable)

5 bytes

Unions

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

Memory Sharing

X & Y

e (union variable)

4 bytes

allocates storage
equal to largest one
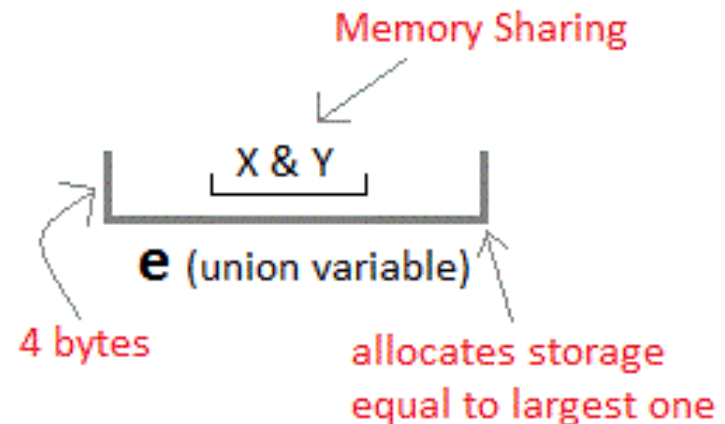
Enumeration (or enum) in C

□ Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

□ Example:

enum State {Working = 1, Failed = 0};

Enumeration (or enum) in C

Example:

enum flag{constant1, constant2, constant3, ... };

The name of enumeration is "flag" and the constant are the values of the flag. By default, the values of the constants are as follows: constant1 = 0, constant2 = 1, constant3 = 2 and so on.

Enumeration

/* In both of the below cases, "day" is defined as the variable of type week. */

enum week{Mon, Tue, Wed};

enum week day;

/* Or */

enum week{Mon, Tue, Wed} day;

/* An example program to demonstrate working of enum in C*/
#include<stdio.h>
 enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
 int main(){
    enum week day;
    day = Wed;
   printf("%d",day);
    return 0;
}
Output: 2

Pointers

A *pointer* is a reference to another variable (memory location) in a program

- Used to change variables inside a function (reference parameters)
- Used to remember a particular member of a group (such as an array)
- Used in dynamic (on-the-fly) memory allocation (especially of arrays)
- Used in building complex data structures (linked lists, stacks, queues, trees, etc.)

Pointer Notation

□ Consider the declaration,

□ *int* i = 3 ;

□ This declaration tells the C compiler to:

(a) Reserve space in memory to hold the integer value.

(b) Associate the name **i** with this memory location.

(c) Store the value 3 at this location.
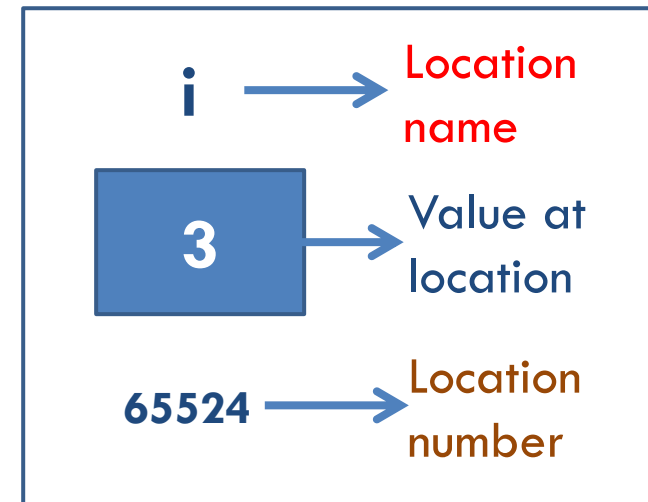
□ The important point is, **i's address in memory is a number**.

i → Location name

3 → Value at location

65524 → Location number

Declaration of Pointer Variable

- General syntax of pointer declaration is

  - **data-type** *****pointer_name**;

- Data type of a pointer must be same as the data type of a variable to which the pointer variable is pointing.

- Void type pointer works with all data types, but is not used often used.

Initialization of Pointer Variable

☐ Pointer Initialization is the process of assigning address of a variable to pointer variable.

☐ Pointer variable contains address of variable of same data type.

☐ In C language **Address Operator &** is used to determine the address of a variable.

☐ The **&** (immediately preceding a variable name) returns the address of the variable associated with it.

Example

int a = 10 ;

int *ptr ;   //pointer declaration

ptr = &a ;   //pointer initialization

or

int *ptr = &a ;   //initialization and

           declaration together

Example(Continue)

main()

{

int i = 3;

printf ("\n Address of I = %u", &i);

printf ("\n Value of I = %d", i);

}

OUTPUT:

Address of i = 65524

Value of i = 3

%**u**, is a format specifier for printing an unsigned integer

'&' used in this statement is C's 'address of' operator.
The expression **&i** returns the address of the variable **i**.

Example (Continue)

main()

{

int i = 3;

printf ("\n Address of i = %u", &i);

printf ("\n Value of i = %d", i);

*printf ("\n Value of i = %d", *(&i));*

}

OUTPUT:

Address of i = 65524

Value of i = 3

Value of i = 3

- The other pointer operator available in C is '*****', called 'value at address' operator.

- It gives the value stored at a particular address.

- The 'value at address' operator is also called 'indirection' operator.

Example (Continue)

```
main()
{
int i = 3;
int *j;
j = &i;

printf ("\n Address of i = %u", &i );
printf ("\n Address of i = %u",j );
printf ("\n Address of j = %u", &j );
printf ("\n Value of j= %u", j);
printf ("\n Value of i = %d", i);
printf ("\n Value of i = %d", *(&i));
printf ("\n Value of i = %d", *j);
}
```


INPUT

OUTPUT:

Address of i = 65524

Address of i = 65524

Address of j = 65522

Value of j = 65524

Value of i = 3

Value of i = 3

Value of i = 3

i is an ordinary **int**,
j is a pointer to an **int** (often called an integer pointer), whereas **k** is a pointer to an integer pointer

```
main()
{
int i = 3, *j, **k;
j = &i;
k = &j;

printf ("\n 1.Address of i = %u", &i );
printf ("\n 2.Address of i = %u", j );
printf ("\n 3.Address of i = %u", *k );
printf ("\n 4.Address of j = %u", &j );
printf ("\n 5.Address of j = %u", k );
printf ("\n 6.Address of k = %u", &k );
printf ("\n 7.Value of j= %u", j);
printf ("\n 8.Value of k= %u", k);
```

```
printf ("\n 9.Value of i = %d", i);
printf ("\n 10.Value of i = %d", *(&i));
printf ("\n 11.Value of i = %d", *j);
printf ("\n 12.Value of i= %d", **k);
}
```

OUTPUT:

i   j   k

1.Address of i = 65524   7.Value of j = 65524

2.Address of i = 65524   8.Value of k = 65522

3.Address of i = 65524   9.Value of i = 3

4.Address of j = 65522   10.Value of i = 3

5.Address of j = 65522   11.Value of i = 3

6.Address of k = 65520   12.Value of i = 3

65524   65522   65520

Pointer Arithmetic

```
main( )
{
    int *ptr;
    char *ptr1;
    float *ptr2;
    ptr++;
ptr1++;
    ptr2++;
    }
```

Output :
ptr=1004
ptr1= 1001
ptr2 = 1008

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

- You can perform arithmetic operations on a pointer just as you can on a numeric value.

- There are four arithmetic operators that can be used on pointers: ++, --, +, and –

- Consider that ptr is an pointer variable which points to the address 1000.

   (Assuming 32-bit integers), Perform the following arithmetic operation on the pointer

Pointer Arithmetic (Continue)

- Consider that ptr is an pointer variable which points to the address 800. (Assuming 32-bit integers), Perform the following arithmetic operation on the pointer −

```
main()
    {   int *ptr;
        char *ptr1;
        float *ptr2;
        ptr- -;    ptr1- - ;    ptr2 - -;
    }
```

Output :
ptr=776
ptr1= 779
ptr2 = 772

- Addition of a number to a pointer
int i =4, *j, *k;
j=&i;
j= j+1;
j= j+9;
K = j+3;

Pointer Arithmetic (Continue)

- Subtraction of a number to a pointer

    int i =4, *j, *k;
    j=&i;
    j= j-2;
    j= j - 5;
    K = j - 3;

- Following operations are not work out on pointers:

  - Additions of 2 pointers

  - Multiplying a pointer with a number

  - Dividing a pointer with a number

Function Call

□ The two types of function calls—

    ◘ **Call by Value**

    ◘ **Call by Reference.**

□ Arguments can generally be passed to functions in one of the two ways:

    ◘ sending the values of the arguments

    ◘ sending the addresses of the arguments

Call by Value Example

```
main()
{
int a = 10, b=20;
swapv(a,b);
printf( "\n a = %d b= %d", a,b);
}
swapv (int x, int y)
{
int t;
t = x;
x = y;
y = t;
printf( "\n x = %d y = %d", x,y);
}
```

OUTPUT:
x = 20   y = 10
a = 10   b = 20

- In this method the 'value' of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function.

- With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function.

Call by Reference Example

```
main()
{
int a = 10, b=20;
swapr(&a,&b);
printf("\n a = %d b= %d", a,b);
}
swapr (int *x, int *y)
{
int t;
t = *x;
*x = *y;
*y = t;
printf("\n x = %d y = %d", *x,*y);
}
```

OUTPUT:

x = 20   y = 10

a = 20   b = 10

- In this method (call by reference) the addresses of actual arguments in the calling function are copied into formal arguments of the called function.

- This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them

Dynamic Memory Allocation

- ☐ Dynamic memory allocation is used to obtain and release memory during program execution

- ☐ Up until this point we reserved memory at compile time using declarations

- ☐ To use the functions discussed here, you must include the **stdlib.h** header file.

- ☐ Four Dynamic Memory Allocation Functions:
  - ▪ Allocate memory - malloc(), calloc(), and realloc()
  - ▪ Free memory - free()

malloc()

To allocate memory use

```
void *malloc(size_t size);
```

- □ Takes number of bytes to allocate as argument
- □ Use sizeof to determine the size of a type
- □ Returns pointer of type void *. A void pointer may be assigned to any pointer
- □ If no memory available, returns NULL

e.g.

```
char *line;
int linelength = 100;
line = (char*)malloc(linelength);
```

malloc() Example

To allocate space for 100 integers:

```
int *ip;

if ((ip = (int*)malloc(100 *
  sizeof(int))) == NULL){
  printf("out of memory\n");
  exit();
  }
```

free()

To release allocated memory use

```
free()
```

- Deallocates memory allocated by malloc()
- Takes a pointer as an argument

e.g.
```
free(newPtr);
```

- Freeing unused memory is a good idea, but it's not mandatory
- When your program exits, any memory which it has allocated but not freed will be automatically released

calloc()

□ Similar to malloc(), the main difference is that the values stored in the allocated memory space are zero by default. With malloc(), the allocated memory could have any value.

□ calloc() requires two arguments - the number of variables you'd like to allocate memory for and the size of each variable.

```
void *calloc(size_t nitem, size_t
    size);
```

calloc() Example

```c
/* Using calloc() to initialize 100 floats to 0.0 */
#include <stdlib.h>
#include <stdio.h>
#define BUFFER_SIZE 100

int main(){
float * buffer;
int i;

if ((buffer = (float*)calloc(BUFFER_SIZE, sizeof(float))) ==
   NULL){
   printf("out of memory\n");
   exit(1);
   }

for (i=0; i < BUFFER_SIZE; i++)
   printf("buffer[%d] = %f\n", i, buffer[i]);

return 0;}
```

What is a File ?

- A *file* is a collection of related data that a computers treats as a single unit.

- Computers store files to secondary storage so that the contents of files remain intact when a computer shuts down.

- When a computer reads a file, it copies the file from the storage device to memory; when it writes to a file, it transfers data from memory to the storage device.

- C uses a structure called **FILE** (defined in **stdio.h**) to store the attributes of a file.

Steps in Processing a File

- Create the stream via a pointer variable using the **FILE** structure:
  FILE *p;

- Open the file, associating the stream name with the file name.

- Read or write the data.

- Close the file.

The Basic File Operations

- **fopen –** open a file- specify how its opened (read/write) and type (binary/text)

- **fclose –** close an opened file

- **fread –** read from a file

- **fwrite –** write to a file

- **fseek/fsetpos –** move a file pointer to somewhere in a file

- **ftell/fgetpos –** tell you where the file pointer is located

File Open Modes

Mode	Meaning
r	Open text file in read mode • If file exists, the marker is positioned at beginning. • If file doesn't exist, error returned.
w	Open text file in write mode • If file exists, it is erased. • If file doesn't exist, it is created.
a	Open text file in append mode • If file exists, the marker is positioned at end. • If file doesn't exist, it is created.

File Open Modes

More on File Open Modes

Mode

r — Open existing file for reading

EOF

File marker positioned at beginning of file

(a) Read Mode

Mode

w — Open new file for writing

EOF

File marker positioned at beginning of file

(b) Write Mode

Mode

a — Open existing file for writing or create new file

EOF

File marker positioned at end of file

(c) Append Mode

File Open Modes

More on File Open Modes (Cont.)

- **r+ :** open for reading and writing, start at beginning

- **w+ :** open for reading and writing (overwrite file)

- **a+ :** open for reading and writing (append if file exists)

File Open

□ The file open function (`fopen`) serves two purposes:

    ◘ It makes the connection between the physical file and the stream.

    ◘ It creates "a program file structure to store the information" C needs to process the file.

□ Syntax:
filepointer=**fopen("filename",**
                              "mode");)

More on fopen

☐ The file mode tells C how the program will use the file.

☐ The filename indicates the system name and location for the file.

☐ We assign the return value of **fopen** to our pointer variable:
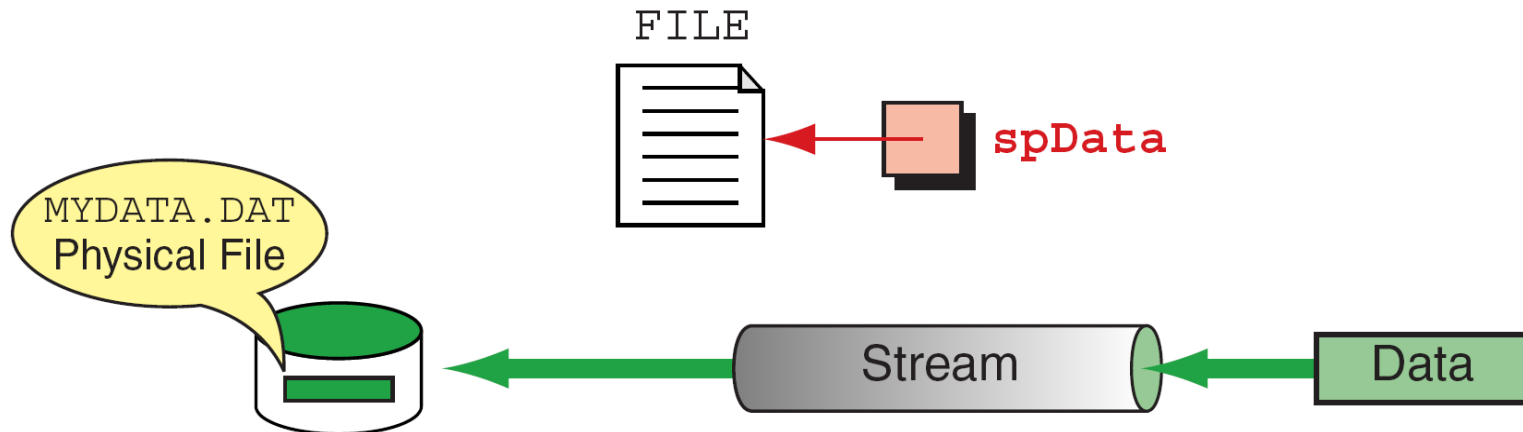
   ▫ spData = fopen("MYFILE.TXT", "w");

More on fopen

from Figure 7-3 in **Forouzan & Gilberg,** *p. 399*

Closing a File

☐ When we finish with a mode, we need to close the file before ending the program or beginning another mode with that same file.

☐ To close a file, we use *fclose* and the pointer variable:

   ◻ **fclose(spData);**

fprintf()

Syntax:

   fprintf (fp,"string",variables);

Example:

```
int i = 12;
float x = 2.356;
char ch = 's';
FILE *fp;
fp=fopen("out.txt","w");
fprintf (fp, "%d %f %c", i, x, ch);
```

fscanf()

Syntax:

   fscanf (fp,"string",identifiers);

Example:

FILE *fp;

Fp=fopen("input.txt","r");

int i;

fscanf (fp,"%d",i);

getc()

Syntax:

identifier = getc (file pointer);

Example:

FILE *fp;

fp=fopen("input.txt","r");

char ch;

ch = getc (fp);

putc()

Write a single character to the output file,
 pointed to by fp.

Example:

FILE *fp;

char ch;

putc (ch,fp);

fread ()

Declaration:

   size_t fread(void *ptr, size_t size, size_t n, FILE *stream);

 Remarks:

☐ fread reads a specified number of equal-sized data items from an input stream into a block.

 ptr   = Points to a block into which data is read
 size   = Length of each item read, in bytes
 n   = Number of items read
 stream = file pointer

Example

Example:

```c
#include <stdio.h>
int main()
{
    FILE *f;
    char buffer[11];
    if (f = fopen("fred.txt", "r"))
    {
    fread(buffer, 1, 10, f);
    buffer[10] = 0;
    fclose(f);
    printf("first 10 characters of the file:\n%s\n", buffer);
    }
 return 0; }
```

fwrite()

Declaration:

size_t fwrite(const void *ptr, size_t size, size_t n, FILE*stream);

Remarks:

☐ fwrite appends a specified number of equal-sized data items to an output file.

ptr   = Pointer to any object; the data written begins at ptr

size   = Length of each item of data

n   =Number of data items to be appended

stream = file pointer

Example

```c
#include <stdio.h>
int main()
{
    char a[10]={'1','2','3','4','5','6','7','8','9','a'};
    FILE *fs;
    fs=fopen("Project.txt","w");
    fwrite(a,1,10,fs);
    fclose(fs);
    return 0;
}
```

fseek()

□ This function sets the file position indicator for the stream pointed to by stream or you can say it seeks a specified place within a file and modify it.

SEEK_SET   **Seeks from beginning of file**

SEEK_CUR   **Seeks from current position**

SEEK_END   **Seeks from end of file**

Example:

```c
#include <stdio.h>
int main()
{
        FILE * f;
        f = fopen("myfile.txt", "w");
        fputs("Hello World", f);
        fseek(f, 6, SEEK_SET);          SEEK_CUR,   SEEK_END
        fputs(" India", f);
        fclose(f);
        return 0;    }
```

End of File

☐ There are a number of ways to test for the end-of-file condition. Another way is to use the value returned by the *fscanf* function:

```
FILE *fptr1;
int istatus ;
istatus = fscanf (fptr1, "%d", &var) ;
if ( istatus == feof(fptr1) )
{
        printf ("End-of-file encountered.\n") ;
}
```