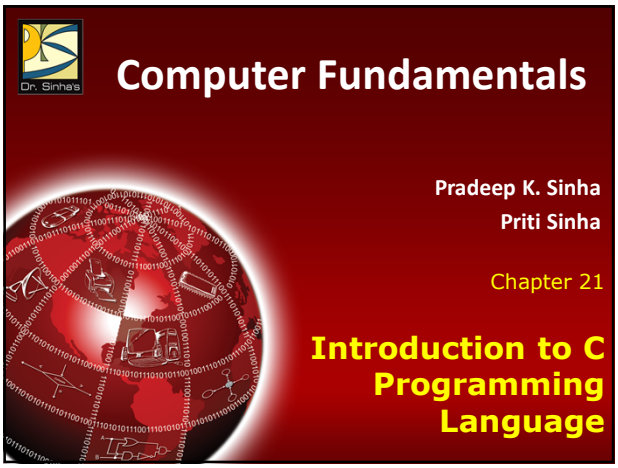


Learning Objectives

In this chapter you will learn about:

- Features of C
- Various constructs and their syntax
- Data types and operators in C
- Control and Loop Structures in C
- Functions in C
- Writing programs in C

Features of C

- Reliable, simple, and easy to use
- Has virtues of high-level programming language with efficiency of assembly language
- Supports user-defined data types
- Supports modular and structured programming concepts
- Supports a rich library of functions
- Supports pointers with pointer operations
- Supports low-level memory and device access
- Small and concise language
- Standardized by several international standards body

C Character Set

Category	Valid Characters	Total
Uppercase alphabets	A, B, C, ..., Z	26
Lowercase alphabets	a, b, c, ..., z	26
Digits	0, 1, 2, ..., 9	10
Special characters	~ `! @ # % ^ & * ( ) _ – + = | \ { } [ ] : ; " ' < > , . ? /	31
		93

Constants

- Constant is a value that never changes
- Three primitive types of constants supported in C are:
  - Integer
  - Real
  - Character

Rules for Constructing Integer Constants

- Must have at least one digit
- + or – sign is optional
- No special characters (other than + and – sign) are allowed
- Allowable range is:
  - -32768 to 32767 for integer and short integer constants (16 bits storage)
  - -2147483648 to 2147483647 for long integer constants (32 bits storage)
- Examples are: 8, +17, -6

Rules for Constructing Real Constants in Fractional Form

- Must have at least one digit
- Must have one and only one decimal point
- + or – sign is optional
- No special characters (other than + and – sign) are allowed
- Examples are: 5.3, +18.59, -0.46

Rules for Constructing Real Constants in Exponential Form

- Has two parts – mantissa and exponent - separated by 'e' or 'E'
- Mantissa part is constructed by the rules for constructing real constants in fractional form
- Exponent part is constructed by the rules for constructing integer constants
- Allowable range is -3.4e38 to 3.4e38
- Examples are: 8.6e5, +4.3E-8, -0.1e+4

Rules for Constructing Character Constants

- Single character from C character set
- Enclosed within single inverted comma (also called single quote) punctuation mark
- Examples are: 'A' 'a' '8' '%'

Variables

- A C variable is an entity whose value may vary during program execution
- It has a name and type associated with it
- Variable name specifies programmer given name to the memory area allocated to a variable
- Variable type specifies the type of values a variable can contain
- Example: In $i = i + 5$, i is a variable

Rules for Constructing Variable Names

- Can have 1 to 31 characters
- Only alphabets, digits, and underscore (as in *last_name*) characters are allowed
- Names are case sensitive (*nNum* and *nNUM* are different)
- First character must be an alphabet
- Underscore is the only special character allowed
- Keywords cannot be used as variable names
- Examples are: I saving_2007 ArrSum

Data Types Used for Variable Type Declaration

Data Type	Minimum Storage Allocated	Used for Variables that can contain
int	2 bytes (16 bits)	integer constants in the range -32768 to 32767
short	2 bytes (16 bits)	integer constants in the range -32768 to 32767
long	4 bytes (32 bits)	integer constants in the range -2147483648 to 2147483647
float	4 bytes (32 bits)	real constants with minimum 6 decimal digits precision
double	8 bytes (64 bits)	real constants with minimum 10 decimal digits precision
char	1 byte (8 bits)	character constants
enum	2 bytes (16 bits)	Values in the range -32768 to 32767
void	No storage allocated	No value assigned

Variable Type Declaration Examples

```
int       count;
short     index;
long      principle;
float     area;
double    radius;
char      c;
```


Standard Qualifiers in C

Category	Modifier	Description
Lifetime	auto register static extern	Temporary variable Attempt to store in processor register, fast access Permanent, initialized Permanent, initialized but declaration elsewhere
Modifiability	const volatile	Cannot be modified once created May be modified by factors outside program
Sign	signed unsigned	+ or – + only
Size	short long	16 bits 32 bits

Lifetime and Visibility Scopes of Variables

- Lifetime of all variables (except those declared as *static*) is same as that of function or statement block it is declared in
- Lifetime of variables declared in global scope and static is same as that of the program
- Variable is visible and accessible in the function or statement block it is declared in
- Global variables are accessible from anywhere in program
- Variable name must be unique in its visibility scope
- Local variable has access precedence over global variable of same name

Keywords

- *Keywords* (or reserved words) are predefined words whose meanings are known to C compiler
- C has 32 keywords
- Keywords cannot be used as variable names

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Comments

- Comments are enclosed within /* and */
- Comments are ignored by the compiler
- Comment can also split over multiple lines
- Example: /* This is a comment statement */

Operators

- Operators in C are categorized into data access, arithmetic, logical, bitwise, and miscellaneous
- ***Associativity*** defines the order of evaluation when operators of same precedence appear in an expression
  - a = b = c = 15, '=' has R → L associativity
  - First c = 15, then b = c, then a = b is evaluated
- ***Precedence*** defines the order in which calculations involving two or more operators is performed
  - x + y * z , '*' is performed before '+'

Arithmetic Operators

Operator	Meaning with Example	Associativity	Precedence
	Arithmetic Operators		
+	Addition; x + y	L → R	4
-	Subtraction; x - y	L → R	4
*	Multiplication; x * y	L → R	3
/	Division; x / y	L → R	3
%	Remainder (or Modulus); x % y	L → R	3
++	Increment;		
	x++ means post-increment (increment the value of x by 1 after using its value);	L → R	1
	++x means pre-increment (increment the value of x by 1 before using its value)	R → L	2

Arithmetic Operators

Operator	Meaning with Example	Associativity	Precedence
Arithmetic Operators			
--	Decrement;		
	x-- means post-decrement (decrement the value of x by 1 after using its value);	L → R	1
	--x means pre-decrement (decrement the value of x by 1 before using its value)	R → L	2
=	x = y means assign the value of y to x	R → L	14
+=	x += 5 means x = x + 5	R → L	14
-=	x -= 5 means x = x - 5	R → L	14
* =	x *= 5 means x = x * 5	R → L	14
/=	x /= 5 means x = x / 5	R → L	14
%=	x %= 5 means x = x % 5	R → L	14

Logical Operators

Operator	Meaning with Example	Associativity	Precedence
Logical Operators			
!	Reverse the logical value of a single variable; !x means if the value of x is non-zero, make it zero; and if it is zero, make it one	R → L	2
>	Greater than; x > y	L → R	6
<	Less than; x < y	L → R	6
>=	Greater than or equal to; x >= y	L → R	6
<=	Less than or equal to; x <= y	L → R	6
==	Equal to; x == y	L → R	7
!=	Not equal to; x != y	L → R	7
&&	AND; x && y means both x and y should be true (non-zero) for result to be true	L → R	11
||	OR; x || y means either x or y should be true (non-zero) for result to be true	L → R	12
z?x:y	If z is true (non-zero), then the value returned is x, otherwise the value returned is y	R → L	13

Bitwise Operators

Operator	Meaning with Example	Associativity	Precedence
Bitwise Operators			
~	Complement; ~x means All 1s are changed to 0s and 0s to 1s	R → L	2
&	AND; x & y means x AND y	L → R	8
|	OR; x | y means x OR y	L → R	10
^	Exclusive OR; x ^ y means x or y	L → R	9
<<	Left shift; x << 4 means shift all bits in x four places to the left	L → R	5
>>	Right shift; x >> 3 means shift all bits in x three places to the right	L → R	5
&=	x &= y means x = x & y	R → L	14
|=	x |= y means x = x | y	R → L	14
^=	x ^= y means x = x ^ y	R → L	14
<<=	x <<= 4 means shift all bits in x four places to the left and assign the result to x	R → L	14
>>=	x >>= 3 means shift all bits in x three places to the right and assign the result to x	R → L	14

Data Access Operators

Operator	Meaning with Example	Associativity	Precedence
Data Access Operators			
x[y]	Access y^{th} element of array x; y starts from zero and increases monotically up to one less than declared size of array	L → R	1
x.y	Access the member variable y of structure x	L → R	1
x –›y	Access the member variable y of structure x	L → R	1
&x	Access the address of variable x	R → L	2
*x	Access the value stored in the storage location (address) pointed to by pointer variable x	R → L	2

Miscellaneous Operators

Operator	Meaning with Example	Associativity	Precedence
Miscellaneous Operators			
x(y)	Evaluates function x with argument y	L → R	1
sizeof (x)	Evaluate the size of variable x in bytes	R → L	2
sizeof (type)	Evaluate the size of data type "type" in bytes	R → L	2
(type) x	Return the value of x after converting it from declared data type of variable x to the new data type "type"	R → L	2
x,y	Sequential operator (x then y)	L → R	15

Statements

Statements

- C program is a combination of statements written between { and } braces
- Each statement performs a set of operations
- Null statement, represented by ";" or empty {} braces, does not perform any operation
- A simple statement is terminated by a semicolon ";"
- Compound statements, called *statement block*, perform complex operations combining null, simple, and other block statements

Examples of Statements

- a = (x + y) * 10; /* simple statement */
- if (sell > cost) /* compound statement follows */

```
{
  profit = sell – cost;
  printf ("profit is %d", profit);
}
else      /* null statement follows */
{
}
```


Simple I/O Operations

- C has no keywords for I/O operations
- Provides standard library functions for performing all I/O operations

Basic Library Functions for I/O Operations

I/O Library Functions	Meanings
getch()	Inputs a single character (most recently typed) from standard input (usually console).
getche()	Inputs a single character from console and echoes (displays) it.
getchar()	Inputs a single character from console and echoes it, but requires *Enter* key to be typed after the character.
putchar() or putch()	Outputs a single character on console (screen).
scanf()	Enables input of formatted data from console (keyboard). Formatted input data means we can specify the data type expected as input. Format specifiers for different data types are given in Figure 21.6.
printf()	Enables obtaining an output in a form specified by programmer (formatted output). Format specifiers are given in Figure 21.6. Newline character "\n" is used in *printf()* to get the output split over separate lines.
gets()	Enables input of a string from keyboard. Spaces are accepted as part of the input string, and the input string is terminated when *Enter* key is hit. Note that although *scanf()* enables input of a string of characters, it does not accept multi-word strings (spaces in-between).
puts()	Enables output of a multi-word string

Basic Format Specifiers for *scanf()* and *printf()*

Format Specifiers	Data Types
%d	integer (short signed)
%u	integer (short unsigned)
%ld	integer (long signed)
%lu	integer (long unsigned)
%f	real (float)
%lf	real (double)
%c	character
%s	string

Formatted I/O Example

```
/* A portion of C program to illustrate formatted input and output */

        int maths, science, english, total;
        float percent;

        clrscr();                       /* A C library function to make the screen clear  */
        printf ( "Maths marks = " );    /* Displays "Maths marks = "  */
        scanf ( "%d", &maths);          /* Accepts entered value and stores in variable "maths"  */
        printf ( "\n Science marks = " ); /* Displays "Science marks = " on next line because of \n  */
        scanf ( "%d", &science);        /* Accepts entered value and stores in variable "science"  */
        printf ( "\n English marks = " ); /* Displays "English marks = "  on next line because of \n  */
        scanf ( "%d", &english);        /* Accepts entered value and stores in variable "english"  */

        total = maths + science + english;
        percent = total/3;              /* Calculates percentage and stores in variable "percent"  */

         printf ( "\n Percentage marks obtained = %f", percent);
                                        /* Displays "Percentage marks obtained = 85.66" on next line
                                           because of \n  */
```

(Continued on next slide)

Formatted I/O Example

Output:

```
Maths marks = 92
Science marks = 87
English marks = 78
Percentage marks obtained = 85.66
```


Preprocessor Directives

Preprocessor Directives

- *Preprocessor* is a program that prepares a program for the C compiler
- Examples of some common preprocessor directives in C are:

Preprocessor directive	Use
#include	Used to look for a file and place its contents at the location where this preprocessor directives is used
#define	Used for macro expansion
#ifdef..#endif	Used for conditional compilation of segments of a program

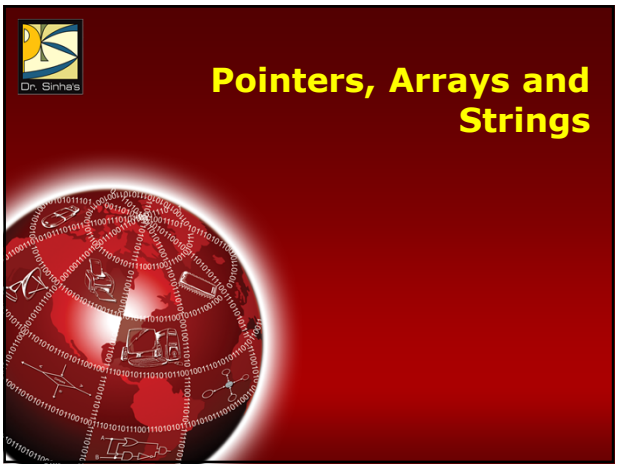
Examples of Preprocessor Directives

```
#include <stdio.h>
#define PI 3.1415
#define AND   &&
#define ADMIT          printf ("The candidate can be admitted");

        #ifdef WINDOWS
                        .
                        .
                        .
            Code specific to windows operating system
                        .
                        .
                        .
        #else
                        .
                        .
                        .
          Code specific to Linux operating system
                        .
                        .
                        .
        #endif
                        .
                        .
                        .
          Code common to both operating systems
```


Standard Preprocessor Directives in C

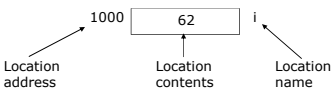
Preprocessor Directive	Meaning	Category
#	Null directive	Simple
#error *message*	Prints *message* when processed	
#line *linenum filename*	Used to update code line number and filename	
#pragma *name*	Compiler specific settings	
#include *filename*	Includes content of another file	File
#define *macro/string*	Define a macro or string substitution	Macro
#undef macro	Removes a macro definition	
#if *expr*	Includes following lines if *expr* is true	Conditional
# elif *expr*	Includes following lines if *expr* is true	
#else	Handles otherwise conditions of #if	
#endif	Closes #if or #elif block	
#ifdef *macro*	Includes following lines if macro is defined	
#ifndef imacro	Includes following lines if macro is not defined	
#	String forming operator	Operators
##	Token pasting operator	
defined	same as *#ifdef*	



Pointers

- C pointers allow programmers to directly access memory addresses where variables are stored
- Pointer variable is declared by adding a '∗' symbol before the variable name while declaring it.
- If *p* is a pointer to a variable (e.g. int i, *p = i;)
  - Using *p* means address of the storage location of the pointed variable
  - Using ∗*p* means value stored in the storage location of the pointed variable
- Operator '&' is used with a variable to mean variable's address, e.g. &i gives address of variable i

Illustrating Pointers Concept



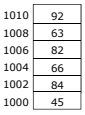
Address of i = 1000
Value of i = 62

```
int i = 62;
int *p;
int j;
p = &i;          /* p becomes 1000 */
j = *p;          /* j becomes 62 */
j = 0;           /* j becomes zero */
j = *(&i)        /* j becomes 62 */
```

Array

- An array is a collection of fixed number of elements in which all elements are of the same data type
- It is a homogeneous, linear, and contiguous memory structure
- Its elements can be referred to by using their subscript or index position that is monotonic in nature
- First element is always denoted by subscript value of 0 (zero), increasing monotonically up to one less than declared size of array
- Before using an array, its type and dimension must be declared
- An array can also be declared as multi-dimensional such as Matrix2D[10][10]

Illustrating Arrays Concept



int marks[6];

Each element being an int occupies 2 bytes

marks[0] = 45
marks[1] = 84
.
.
.
marks[5] = 92

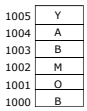
(a) An array of integers having 6 elements

float price[4];

Each element being a float occupies 4 bytes

price[0] = 82.75
price[1] = 155.50
.
.
.
price[3] = 10.25

(b) An array of real numbers having 4 elements



char city[6];

Each element being a char occupies 1 byte

city[0] = 'B'
city[1] = 'O'
.
.
.
city[5] = 'Y'

(c) An array of characters having 6 elements

String

- A string is a one-dimensional array of characters terminated by a null character ('\0')
- It is initialized at declaration as

  char name[] = "PRADEEP";
- Its individual elements can be accessed in the same way as we access array elements such as name[3] = 'D'
- Strings are used for text processing
- C provides a rich set of string handling library functions

Library Functions for String Handling

Library Function	Used To
strlen	Obtain the length of a string
strlwr	Convert all characters of a string to lowercase
strupr	Convert all characters of a string to uppercase
strcat	Concatenate (append) one string at the end of another
strncat	Concatenate only first n characters of a string at the end of another
strcpy	Copy a string into another
strncpy	Copy only the first n characters of a string into another
strcmp	Compare two strings
strncmp	Compare only first n characters of two strings
stricmp	Compare two strings without regard to case
strnicmp	Compare only first n characters of two strings without regard to case
strdup	Duplicate a string
strchr	Find first occurrence of a given character in a string
strrchr	Find last occurrence of a given character in a string
strstr	Find first occurrence of a given string in another string
strset	Set all characters of a string to a given character
strnset	Set first n characters of a string to a given character
strrev	Reverse a string

User Defined Data Types

User Defined Data Types (UDTs)

- UDT is composite data type whose composition is not included in language specification
- Programmer declares them in a program where they are used
- Two types of UDTs are:
  - Structure
  - Union

Structure

- It is a UDT containing a number of data types grouped together
- Its constituents data types may or may not be of different types
- It has continuous memory allocation and its minimum size is the sum of sizes of its constituent data types
- All elements (member variable) of a structure are publicly accessible
- Each member variable can be accessed using "." (dot) operator or pointer (*EmpRecord.EmpID* or *EmpRecord → EmpID*)
- It can have a pointer member variable of its own type, which is useful in crating linked list and similar data structures

Structure (Examples)

```
struct Employee
{
        int EmpID;
        char EmpName[20];
};
```

```
struct Employee
{
    int  EmpID;
    char EmpName[20];
} EmpRecord;
```

```
Struct Employee EmpRecord;
Struct Employee *pempRecord = &EmpRecord;
```


Union

- It is a UDT referring to same memory location using several data types
- It is a mathematical union of all constituent data types
- Each data member begins at the same memory location
- Minimum size of a union variable is the size of its largest constituent data types
- Each member variable can be accessed using "," (dot) operator
- Section of memory can be treated as a variable of one type on one occasion, and of another type on another occasion

Union Example

```
unionNum
{
        int intNum;
        unsigned
unsNum'
};
union Num Number;
```


Difference Between Structure and Union

- Both group a number of data types together
- Structure allocates different memory space contiguously to different data types in the group
- Union allocates the same memory space to different data types in the group

Control Structures

Control Structures

- *Control structures* (branch statements) are decision points that control the flow of program execution based on:
  - Some condition test (conditional branch)
  - Without condition test (unconditional branch)
- They ensure execution of other statement/block or cause skipping of some statement/block

Conditional Branch Statements

- **if** is used to implement simple one-way test. It can be in one of the following forms:
  - if..stmt
  - if..stmt1..else..stmt2
  - if..stmt1..else..if..stmtn
- **switch** facilitates multi-way condition test and is very similar to the third *if* construct when primary test object remains same across all condition tests

Examples of "if" Construct

-
```
if (i <= 0)
    i++;
```
-
```
if (i <= 0)
    i++;
else
  j++;
```
-
```
if (i <= 0)
    i++;
else if (i >= 0)
    j++;
else
  k++;
```

Example of "switch" Construct

```
switch(ch)
{
        case 'A':
        case 'B':
        case 'C':
                printf("Capital");
                break;
        case 'a':
        case 'b':
        case 'c':
                printf("Small");
                break;
        default:
                printf("Not cap or small");
}
```

Same thing can be written also using *if* construct as:

```
if (ch == 'A' || ch == 'B' || ch == 'C')
    printf("Capital");
else if (ch == 'a' || ch == 'b' || ch == 'c')
    printf("Small");
else
    printf("Not cap or small");
```

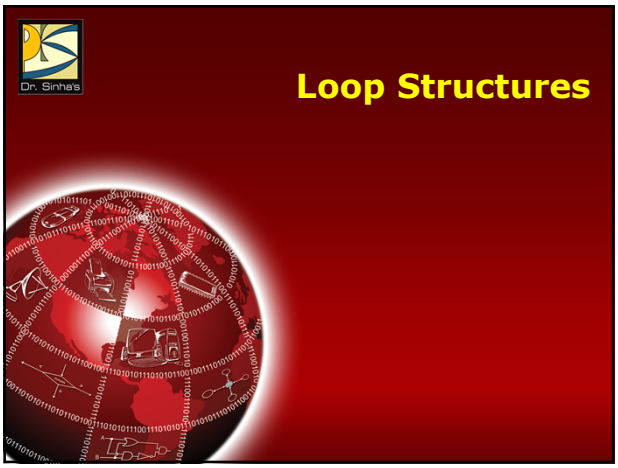

Unconditional Branch Statements

- Break: Causes unconditional exit from *for*, *while*, *do*, or *switch* constructs. Control is transferred to the statement immediately outside the block in which *break* appears.
- Continue: Causes unconditional transfer to next iteration in a *for*, *while*, or *do* construct. Control is transferred to the statement beginning the block in which *continue* appears.
- Goto label: Causes unconditional transfer to statement marked with the label within the function.

(Continued on next slide)

Unconditional Branch Statements

- Return [value/variable]: Causes immediate termination of function in which it appears and transfers control to the statement that called the function. Optionally, it provides a value compatible to the function's return data type.



Loop Structures

- Loop statements are used to repeat the execution of statement or blocks
- Two types of loop structures are:
  - **Pretest**: Condition is tested before each iteration to check if loop should occur
  - **Posttest**: Condition is tested after each iteration to check if loop should continue (at least, a single iteration occurs)

Pretest Loop Structures

- **for**: It has three parts:
  - *Initializer* is executed at start of loop
  - *Loop condition* is tested before iteration to decide whether to continue or terminate the loop
  - *Incrementor* is executed at the end of each iteration
- **While**: It has a *loop condition* only that is tested before each iteration to decide whether to continue or terminate the loop

Examples of "for" and "while" Constructs

- for (i=0; i < 10; i++)
  printf("i = %d", i);

- while (i < 10)
  {
  printf("i = %d", i);
  i++;
  }

Posttest Loop Construct "do...while"

- It has a loop condition only that is tested after each iteration to decide whether to continue with next iteration or terminate the loop
- Example of *do…while* is:

```
do {
    printf("i = %d", i);
    i++;
} while (i < 10) ;
```


Functions

- Functions (or subprograms) are building blocks of a program
- All functions must be declared and defined before use
- Function declaration requires *function name*, *argument list*, and *return type*
- Function definition requires coding the body or logic of function
- Every C program must have a *main* function. It is the entry point of the program

Example of a Function

```
int myfunc ( int Val, int ModVal )
{
    unsigned temp;
    temp = Val % ModVal;
    return temp;
}
```

This function can be called from any other place using simple statement:

```
int n = myfunc(4, 2);
```


Sample C Program (Program-1)

```
/* Program to accept an integer from console and to display
   whether the number is even or odd */

# include <stdio.h>
void main()
{
        int number, remainder;
        clrscr(); /* clears the console screen */
        printf ("Enter an integer:   ");
        scanf ("%d", &number);
        remainder = number % 2;
        if (remainder == 0)
                printf ("\n   The given number is even");
        else
                printf ("\n   The given number is odd");

        getch();
}
```

Sample C Program (Program-2)

```
/* Program to accept an integer in the range 1 to 7 (both inclusive) from
   console and to display the corresponding day (Monday for 1, Tuesday for
   2, Wednesday for 3, and so on). If the entered number is out of range,
   the program displays a message saying that */

# include <stdio.h>
# include <conio.h>

#define MON printf ("\n   Entered number is 1 hence day is MONDAY");
#define TUE printf ("\n   Entered number is 2 hence day is TUESDAY");
#define WED printf ("\n   Entered number is 3 hence day is WEDNESDAY");
#define THU printf ("\n   Entered number is 4 hence day is THURSDAY");
#define FRI printf ("\n   Entered number is 5 hence day is FRIDAY");
#define SAT printf ("\n   Entered number is 6 hence day is SATURDAY");
#define SUN printf ("\n   Entered number is 7 hence day is SUNDAY");
#define OTH printf ("\n   Entered number is out of range");

void main()
{
        int day;
        clrscr();
        printf ("Enter an integer in the range 1 to 7");
        scanf ("%d", &day);
```

(Continued on next slide)

Sample C Program (Program-2)

```
        switch(day)
        {
                Case 1:
                        MON;
                        break;
                Case 2:
                        TUE;
                        break;
                Case 3:
                        WED;
                        break;
                Case 4:
                        THU;
                        break;
                Case 5:
                        FRI;
                        break;
                Case 6:
                        SAT;
                        break;
                Case 7:
                        SUN;
                        break;
                default:
                        OTH;
        }
        getch();
}
```


Sample C Program (Program-3)

```
/* Program to accept the radius of a circle from console and to calculate
   and display its area and circumference */

# include <stdio.h>
# include <conio.h>
# define   PI   3.1415

void main()
{
        float radius, area, circum;
        clrscr();
        printf ("Enter the radius of the circle:    ");
        scanf ("%f", &radius);
        area = PI * radius * radius;
        circum = 2 * PI * radius;
        printf ("\n  Area and circumference of the circle are %f
              and %f respectively", area, circum);
        getch();
}
```

(Continued on next slide)

Sample C Program (Program-4)

```
/* Program to accept a string from console and to display the number of
   vowels in the string */

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

void main()
{
        char   input_string[50];   /* maximum 50 characters */
        int len;
        int   i = 0, cnt = 0;
        clrscr();
        printf ("Enter a string of less than 50 characters: \n");
        gets (input_string);
        len = strlen (input_string);
        for (i = 0;   i < len;   i++)
        {
                switch (input_string[i])
```

(Continued on next slide)

Sample C Program (Program-4)

```
    {
                case 'a':
                case 'e':
                case 'i':
                case 'o':
                case 'u':
                case 'A':
                case 'E':
                case 'I':
                case 'O':
                case 'U':
                        cnt++
            }
        }
        printf ("\n  Number of vowels in the string are: %d", cnt);
        getch();
}
```


Sample C Program (Program-5)

```
/* Program to illustrate use of a user defined function. The program initializes an array of n elements
from 0 to n-1 and then calculates and prints the sum of the array elements. In this example n = 10 */

#include <stdio.h>
#define SIZE 10

int ArrSum(int *p, int n);
{
        int s, tot = 0;
        for(s = 0; s < n; s++)
        {
                tot += *p;
                p++;
        }
        return tot;
}
int main()
{
        int i = 0, sum = 0;
        int nArr[SIZE] = {0};
        while(i < SIZE)
        {
                nArr[i] = i;
                i++
        }
        sum = ArrSum(nArr, SIZE);
        printf("Sum of 0 to 9 = %d\n", sum);
        return 0;
}
```

Key Words/Phrases

- Arithmetic operators
- Arrays
- Assignment operators
- Bit-level manipulation
- Bitwise operators
- Branch statement
- Character set
- Comment statement
- Compound statement
- Conditional branch
- Conditional compilation
- Constants
- Control structures
- Format specifiers
- Formatted I/O
- Function
- Keywords
- Library functions
- Logical operators
- Loop structures
- Macro expansion
- Main function
- Member element
- Null statement
- Operator associativity
- Operator precedence
- Pointer
- Posttest loop
- Preprocessor directives
- Pretest loop
- Primitive data types
- Reserved words
- Simple statement
- Statement block
- Strings
- Structure data type
- Unconditional branch
- Union data type
- User-defined data types
- Variable name
- Variable type declaration
- Variables