Computer Concept and Programming

I semester, BSc. CSIT

UNIT II : Overview of C Language

Compiled by :
Ankit Bhattarai

Unit II

(4 hrs.)

Overview of C Language [4 hrs.]

Introduction, C Character Set, Tokens, Identifiers, Keywords, Constants, Variables, Data Types, Type Conversion, Operators and Expressions, Structure of a C program, Managing Input and Output Operations, Common Errors in Programming, Debugging Basics.

History of C

- **Origins and Precursors (1960s-1970s**): C evolved from BCPL (Basic Combined Programming Language) and B, developed by Martin Richards and Ken Thompson, respectively, as simplified languages for system programming.
- **Creation of C (1972):** Dennis Ritchie at Bell Labs developed C to rewrite the Unix operating system, combining high-level functionality with low-level control, making it portable and efficient.
- **Unix and Adoption (1973-1980):** The Unix OS was rewritten in C, showcasing its portability and performance, leading to widespread adoption in academia, research, and industry.
- **Standardization (1989):** ANSI standardized C as "ANSI C" (C89), followed by international recognition (ISO C). Updates like C99, C11, and C17 introduced modern features, enhancing functionality and safety.
- **Legacy and Impact:** C became the foundation for languages like C++, Java, and Python, and remains pivotal in operating systems, embedded systems, and performance-critical applications.

Basic Structure of C program

Compiled by a

```c
#include <stdio.h>  // Preprocessor directive

// Global declaration
int global_variable = 10;

// Function prototype
void display();

int main() {
    int local_variable = 5;  // Local variable

    printf("Hello, World!\n");
    printf("Local Variable: %d\n", local_variable);
    printf("Global Variable: %d\n", global_variable);

    display();  // Function call

    return 0;  // End of program
}

// User-defined function
void display() {
    printf("This is a user-defined function.\n");
}
```

Basic Structure of C program : Explanation

Preprocessor Directives

These include header files and macros that are processed before compilation.

Example:

```
#include <stdio.h>
#define PI 3.14
```

Global Declarations

Global variables, constants, or function prototypes declared outside of any function.

Example:

```
int global_variable = 10;
void display();  // Function prototype
```

Basic Structure of C program : Explanation

main() Function

Every C program must have a main() function where program execution begins.

Example:

```
int main() {
    // Code starts here
    return 0;
}
```

Local Declarations and Statements

Variables and executable statements inside the main() or other functions.

Example:

```
int main() {
    int local_variable = 5;  // Local variable
    printf("Hello, World!\n");
    return 0;
}
```

Basic Structure of C program : Explanation

User-Defined Functions

Functions created by the programmer for modularity and reusability.

Example:

```
void display() {
    printf("This is a user-defined function.\n");
}
```

Character Set, Token, Keywords, Identifiers

Character Set

- A character set is a collection of valid characters recognized and processed by a compiler or interpreter.
- In the C programming language, characters serve as fundamental units, forming the basis for essential program elements such as constants, variables, and expressions.

The characters available in C are categorized into following types.

1) Letters/Alphabets: Uppercase(A-Z), Lowercase(a-z)

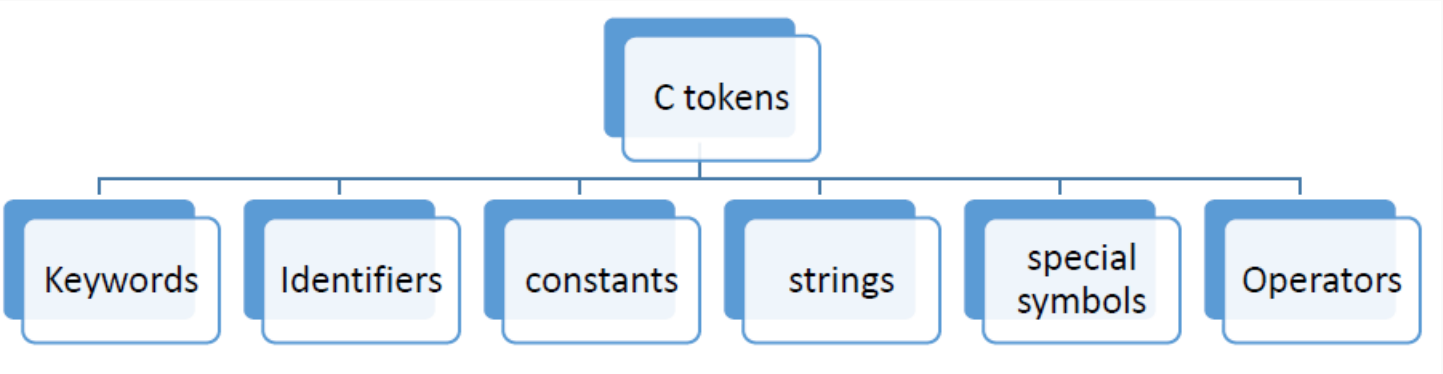
2) Digits (0-9)

3) Special symbols (+,-,*, ! ,%,_,<,>, :)

4) Whitespace characters: Blank Space, Newline, Horizontal tab, Vertical tab

Token

- Tokens are the smallest individual units of program.
- C has six types of tokens .They are as follows:



Keywords

- Keywords are predefined words in the C programming language with reserved meanings for the compiler.
- These keywords have specific functionalities and cannot be used as identifiers (names for variables, functions, etc.
- The standard keywords are:

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

Identifiers

- Identifiers are the names given by user to various program elements such as variables, function and arrays.

Rules of naming Identifiers

- ✓ First character must be alphabet (or underscore).
- ✓ Must consists of only letters, digits or underscore
- ✓ Keyword cannot be used.
- ✓ The length of identifiers should not be greater than 31 characters.
- ✓ Must not contain whitespace.
- ✓ Identifiers are case-sensitive

Data Type

Data Type

- Data type is a classification of type of data that a variable can hold in programming.
- The data type specifies the range of values that can be used and size of memory needed for that value.

Generally data type can be categorized as:

1. **Primary date type** (in next slide)
2. **Derived Data type**: They are derived from existing primary data types. E.g. Array, Union, Structure
3. **User defined data type**: The data type that are defined by user is known as user defined data types.

   Example: enum (Enumerated data type) and typedef (type definition datatype).

   E.g.

   typedef int integer ;

   integer symbolizes int data type Now, we can declare int variable “a” as **integer a** rather than **int a .**

Data Type : Primary data type

- Primary data types are predefined by the C compiler and provide a foundation for creating more complex data structures. The size and range of data types on 16 bit machine are:

Data type	Description	Required Memory	Range	Format specifier
char	Used to store character value Eg. ‘a’,’c’, ‘x’	1 byte	-128 to 127	%c
int	Used to store whole numbers/non fractional numbers. Eg.101, 205, -908	2 bytes	-32768 to +32767	%d
float	Used to store fractional number and has precision of 6 digits. Eg. 5.674, 304.67,67.8903	4 bytes	$-3.4*10^{38}$ to $3.4*10^{38}$	%f
double	Used when precision of float is insufficient and it has precision of 14 digits.	8 bytes	$-1.7*10^{308}$ to $1.7*10^{308}$	%lf
void	Has no values and used as return type for function that do not return a value. Eg. void main()			

Variable

Variable

- A variable in programming is a symbolic name associated with a memory location where data is stored.
- It serves as a container to hold and manipulate data values, which can change dynamically during program execution.

Variable declaration

Naming a variable along with its data type for programming is known as variable declaration.

The declaration deals with two things:

- It tells the compiler what the variable name is
- It specifies what type of data the variable can hold

Syntax:

data_type variable_name ;

eg. int roll;

roll is a variable of type integer. roll can only store integer variable only

Variable

Rules for declaring variables

Variable name must begin with a letter, some system permit underscore as first character.

- ANSI standard recognizes a length of 31 characters.
- It should not be keyword.
- Whitespace is not allowed.
- Variable name are case sensitive .i.e. uppercase and lowercase are different. Thus variable temp is not same as TEMP.
- No two variables of the same name are allowed to be declared in the same scope.

WAP to illustrate variable runtime initialization:


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

int main()
{
int a,b;
printf("Enter two numbers\n");
scanf("%d%d",&a,&b);
printf("Entered numbers are %d and %d",a,b);
getch();
return 0;
}
```

Constants

Constants

1) Numeric Constant

It consist of:

1.1 Integer constant

It refers to sequence of digits.

There are three types of integer constants.

Decimals: Decimal Integer consist of set of digits, 0 through 9, preceded by an optional +ve or –ve sign.eg.124,-321,678, 654343 etc.

Octal: An octal Integer constant consist of any combination of digits from the set 0 to 7, with a leading 0. eg 037, 075, 0664, 0551

Hexadecimal: Hexadecimal are sequence of digits 0-9 and alphabet A-F with preceding 0x or 0X. eg. 0x5567, 0X53A, 0xFF etc.

1.2. Real or floating point constant

They are numeric constant with decimal points. The real constant are further categorized as:

- **Fractional real constant**

They are the set of digits from 0 to 9 with decimal points.eg.394.7867,-0.78676

- **Exponential real constant**

In the exponential form the constants are represented in following form:

mantissa e exponent

where, the mantissa is either a real number expressed in decimal notation or an integer but exponent is always in integer form.

e.g.

21565.32=2.156532E4, Where E4=104

Similarly, -0.000000368 is equivalent to -3.68E-7

2) Character Constant

- **Single Character Constant**

A single character constant (or simply character constant) contains a single character enclosed within a pair of single quote marks. E.g.‘5’ ‘X’ ‘A’ ‘4’ etc.

- **String constant**

A string constant is a sequence of characters enclosed in double quotes. The characters may be numbers, special characters and blank space.eg. “Hello” “green” “305” etc.

Constants : WAP to show different constants in C.


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

// Decimal Integer
int decimalInt = 124;
int negativeDecimalInt = -321;

// Octal Integer (prefixed with 0)
int octalInt1 = 037;
int octalInt2 = 075;

// Hexadecimal Integer (prefixed with 0x or 0X)
int hexInt1 = 0x5567;
int hexInt2 = 0X53A;

// Fractional Real Constant
float fractionalReal1 = 394.7867;
float fractionalReal2 = -0.78676;

// Exponential Real Constant
float exponentialReal1 = 2.156532E4; // Equivalent to 21565.32
```

Constants : WAP to show different constants in C.


```
float exponentialReal2 = -3.68E-7; // Equivalent to -0.000000368

// Single Character Constant
char singleChar1 = '5';
char singleChar2 = 'X';

// String Constant
char stringConstant1[] = "Ambition";
char stringConstant2[] = "College";
char stringConstant3[] = "305";

// Displaying all constants
printf("Numeric Constants:\n");
printf("Decimal Integer: %d, %d\n", decimalInt, negativeDecimalInt);
printf("Octal Integer: %o (Decimal: %d), %o (Decimal: %d)\n", octalInt1, octalInt1,
octalInt2, octalInt2);
printf("Hexadecimal Integer: %x (Decimal: %d), %X (Decimal: %d)\n", hexInt1, hexInt1,
hexInt2, hexInt2);
```

Constants : WAP to show different constants in C.


```
printf("Fractional Real Constants: %.4f, %.5f\n", fractionalReal1, fractionalReal2);
printf("Exponential Real Constants: %.5e, %.5e\n", exponentialReal1, exponentialReal2);
printf("\nCharacter Constants:\n");
printf("Single Character Constants: '%c', '%c'\n", singleChar1, singleChar2);
printf("String Constants: \"%s\", \"%s\", \"%s\"\n", stringConstant1, stringConstant2,
stringConstant3);
return 0;
}
```

Format Specifier

Format Specifier

- Format specifier is used during input and output. It is the way to tell the compiler what type of data is in variable during taking input using scanf() or printing using printf().
- In format specifier the percentage(%) is followed by conversion character. This indicates the corresponding data item.

Example:

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

Here %d is format specifier and it tells the compiler that we want to print an integer value that is present in variable a.

In this way there are several format specifiers in c. Like %c for character, %f for float, etc.

Format Specifier

- List of different format specifiers

Format specifier	Purpose
%c	Character
%d	Signed Integer
%f	Floating point
%u	Unsigned Integer
%o	Octal
%X or %x	Hexadecimal representation of unsigned integer
%e or %E	Scientific Notation of float values
%s	String
%lf	Floating point (double)
%Lf	Floating point (long double)
%%	Prints % character
%n	Prints nothing

Type Casting

Type Casting

Type casting, also known as type conversion, is the process of converting a value of one data type to another. In C, this is achieved using the **cast operator**, which has the following syntax

Syntax:

(type_name) expression

where:

- ✓ **type_name** is the desired data type to which you want to convert the value.
- ✓ **expression** is the value or variable that you want to cast.

Here, it is best practice to convert lower data type to higher data type to avoid data loss.

1. Implicit Conversion:

Implicit conversions do not require any operator for conversion. They are automatically performed when a value is copied to a compatible data type in the program.

Type Casting: Implicit Conversion

*Here, the value of **i** has been promoted from **int** to **float** and we have not had to specify any type-casting operator.*

Code:

```
#include<stdio.h>
int main()
{
int i=20;
float result;
result=i; // implicit
conversion
printf("value=%f", result);
return 0;
}
```

output

value=20.000000

Type Casting: Explicit Conversion

- In this type of conversion one data type is converted forcefully to another data type by the user.

The general rule for cast is

(type-name) expression

Where type-name is one of the standard c data types. The expression may be constant , variable or an expression.

Example	Action
x=(int)7.5	7.5 is converted to integer by truncation
a=(int)21.3/(int)4.5	Evaluated as 21/4 and result would be 5
b=(double)sum/n	Division is done in floating point mode
z=(int)a+b	a is converted into integer and then added to b

Explicit Conversion


```
#include<stdio.h>
#include<conio.h>
int main()
{
int a=5,b=2;
float result;
result=(float)a/b; // Explict conversion
printf("value=%f",result);
getch();
return 0;
}
```

output

value= 2.50000

Delimiters, Escape Sequence, Preprocessor Directive

Delimiters

- A Delimiter is a unique character or series of character that indicates the beginning or end of a specific statement, string or function body set.
- Some examples of delimiters are:

1. **Comma ,** :It is used to separate two or more variables ,constants
2. **Semicolon ;** :It is used to indicate the end of statement
3. **Apostrophes ‘** : It is used to indicate the character constant
4. **Double quotes “** : It is used to indicate the end of string.
5. **White space**: Space, tab, newline etc.
6. **Curly brackets** {}
7. **Round brackets or parentheses** ()

Escape Sequence

- Escape sequences are non-printable characters used for formatting the output only. It is a character combination of backslash (\) followed by a letter or by combination of digits.
- Each sequences are typically used to specify actions such as carriage return, backspace, line feed or move cursors to next line.
- The commonly used Escape sequences are as follows:

Escape Sequence	Purpose
\a	Alert sound .A beep is generated by a computer on execution
\b	Backspace
\n	Newline
\r	carriage return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\”	Display double quote character
\’	Display single quote character
\0	Null character. Marks the end of string

Preprocessor Directives

- Preprocessor directives is a collection of special statement that are executed at the beginning of compilation process. They are placed in a source program before the main function and begins with #(Hash).

Some examples of preprocessor directives are

```
#include<stdio.h> //used for standard input and output
#define PI 3.1416 //used for defining symbolic constant PI as 3.1416
#define TRUE 1 //used for defining True as 1
```

- We use preprocessor directive for **file inclusion, conditional compilation and macro expansion.**

ASCII (American Standard Code for Information Interchange)

ASCII (American Standard Code for Information Interchange)

- ASCII is the American Standard Code for Information Interchange. It defines an encoding standard for the set of characters and unique code for each character in the ASCII chart.
- A character variable holds ASCII value (an integer number between 0 and 127) rather than that character itself in c programming. That value is known as ASCII value.

For example ASCII value of ‘A’ is 65. what this means is that, if you assign ‘A’ to a character variable,65 is stored in that variable rather than ‘A’ itself.

Characters	**ASCII values**
A-Z	65-90
a-z	97-122
0-9	48-57

ASCII (American Standard Code for Information Interchange)

Program to print ASCII value of the given character

```
#include<stdio.h>
#include<conio.h>
int main()
{
char ch;
printf("Enter the character\n");
scanf("%c",&ch);
printf("ASCII value of %c is %d",ch,ch);
getch();
return 0;
}
```

Input/Output Functions in C

Input/output Functions in C

- The C programming language offers a rich set of built-in functions for handling input and output operations.
- These functions are crucial for interacting with the user, reading data from files, and displaying results on the screen.
- The standard input/output header file, <stdio.h>, provides access to these functions.

Input/output functions in C are generally categorized into two primary types:

1. Formatted I/O functions
2. Unformatted I/O functions

Formatted I/O functions: Formatted Input

- **Formatted Input** refers to an input data that can be arranged in a particular format according to user requirements.
- The built-in function scanf() can be used to input data into the computer from standard Input device.

The general form of scanf is

```
scanf("control string",arg1,arg2 ….argn);
```

- The control string refers to field in which data is to be entered.
- The arguments arg1,arg2 ,…argn specify the address of locations where data is stored.

eg. scanf("%d%d",&num1,&num2);

Formatted I/O functions: Formatted Output

- Formatted output functions are used to display data in a particular specified format.
- The printf() is an example of formatted output function.The general form of printf() statement is

```
printf("control string",arg1,arg2 ..... argn);
```

The control string may consist of the following data items.

1. Characters that will be printed on the screen as they appear.
2. Format specifications that define the output format for display of each item.
3. Escape sequence characters such as \n,\t and \b

The arguments arg1,arg2,…..argn are the variables whose value are formatted and printed according to specifications of the control string.

Eg. printf("First number=%d\tSecond number=%d",num1,num2);

Formatted I/O functions

Program to read multiple values of different data types using single scanf() function.

```
#include<stdio.h>
#include<conio.h>
int main()
{
char name[20];
int age;
char gender;
float weight;
printf("Enter your Name,Age,Gender and Weight\n");
scanf("%s %d %c %f",name,&age,&gender,&weight);
printf("Your Name=%s\n",name);
printf("Your Age=%d\n",age);
printf("Your Gender=%c\n",gender);
printf("Your Weight=%0.2f",weight);
getch();
return 0;
}
```

Unformatted I/O functions

- Unformatted I/O function do not allow the user to read or display data in a desired format.
- These type of library functions basically deal with a single character or string of characters. The functions getchar(), putchar(), gets(), puts(), getch(),getche(),putch() are considered as unformatted functions.

i) getchar() and putchar()

The getchar() reads a character from a standard input device. It buffers the input until the ‘ENTER’ key is pressed and then assigns this character to character variable.

Syntax is : `character_variable=getchar();`

where character_variable refers to some previously declared character variable.

putchar() function displays a character to standard output device.

Syntax is: `putchar(character_variable);`

Unformatted I/O functions

Program

```
#include<stdio.h>
#include<conio.h>
int main()
{
char ch;
printf("Enter the character");
ch=getchar();
printf("Entered characer is :");
putchar(ch);
getch();
return 0;
}
```

Unformatted I/O functions

ii) getch(), getche() and putch()

The function getch() and getche() both reads a single character in a instant. It is typed without pressing the ENTER key. The difference between them is that getch() reads a character typed without echoing it on a screen, while getche() reads the character and echoes (displays) it onto the screen.

Syntax:

```
character_variable=getch();
characher_varibale=getche();
```

The function putch() prints a character onto the screen.

Syntax:

```
putch(character_variable);
```

Unformatted I/O functions

Program

```
#include<stdio.h>
#include<conio.h>
int main()
{
char ch1,ch2;
printf("Enter First character:");
ch1=getch();
printf("\nEnter Second character:");
ch2=getche();
printf("\n First character is:");
putch(ch1);
printf("\nSecond character is:");
putch(ch2);
getch();
return 0;
}
```

Output

Enter First character:
Enter Second character: b
First character is: a
Second character is: b

Note:

- At first input getch() function read the character input from the user but that the entered character was not displayed.
- But in second input getche() function read the a character from the user input and entered character was echoed to the user without pressing any key.

Unformatted I/O functions

iii) gets() and puts()

- The gets() function is used to read a string of text containing whitespaces, until newline character is encountered.
- It can be used as an alternative function for reading strings. Unlike scanf() functions, it does not skip whitespaces(ie.It can be used to read multiwords string)

Syntax: `gets(string_variable);`

- The puts() function is used to display the string on the screen.

Syntax: `puts(string_variable);`

Unformatted I/O functions : gets() and puts() C program

Program

```
#include<stdio.h>
#include<conio.h>
int main()
{
char name[20];
printf("Enter your name:\n");
gets(name);
printf("Your name is:");
puts(name);
getch();
return 0;
}
```

Operators

Operators

Operator

- Operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.
- Operators are used in program to manipulate data and variables.

Operand

The data items on which operators are act upon are called operands.

e.g. a + b

- Here, symbol + is known as operator
- a and b are operands

Expression

The combination of variables, constants and operators written according to syntax of programming language is known as Expression. e.g. a + b * c - 5

Operators

Types of Operators are classified as:

1. On the basis of no of operands

2. On the basis of function/utility

1. On the basis of number of operands

i) Unary operator

The operator which operates on single operand known as unary operator.

(++) increment Operator

(--) Decrement operator

+ Unary plus

- Unary minus

eg . ++x, -5, +8 , y- - etc.

Operators

ii) Binary operator

The operator which operates on two operands are known as binary operator. e.g. +(addition), – (subtraction) , /(division) , * (multiplication) , <(less than) >(greater than) etc. E.g 3+3, 5>2 , 6*7 , 15-4

iii) Ternary operators

The Operators that operates on three operands are known as ternary operator.

Ternary operator pair "?:" is available in c .

Syntax:

```
expression1? expression2:expression3
```

Example:
Let us consider the program statement
int a=15,b=10,max;

```
max = (a>b)?a: b;
```

Here, The value of a will assigned to max.

The expression1 is evaluated first,

- If it is true then expression2 is evaluated and its value becomes value of the expression
- If it is false expression3 is evaluated and its value becomes value of the expression

On the basics of utility /functions

1) Arithmetic Operators

They are used to perform Arithmetic/Mathematical operations on operands.

Operator	Meaning	Example If a=20 and b=10
+	Addition	a+b=30
-	Subtraction	a-b=10
*	Multiplication	a*b=200
/	Division	a/b=2
%	Modulo Division	a%b=0

Operators: Arithmetic Operators

WAP to illustrate Arithmetic Operators in C

```c
#include <stdio.h>

int main() {
    int num1, num2;
    int sum, difference, product, remainder;
    float quotient;

    // Input two numbers
    printf("Enter two integers: ");
    scanf("%d %d", &num1, &num2);

    // Arithmetic operations
    sum = num1 + num2;
    difference = num1 - num2;
    product = num1 * num2;
    quotient = (float)num1 / num2;
    remainder = num1 % num2;

    // Display the results
    printf("\nResults of Arithmetic Operations:\n");
    printf("Addition: %d + %d = %d\n", num1, num2, sum);
    printf("Subtraction: %d - %d = %d\n", num1, num2, difference);
    printf("Multiplication: %d * %d = %d\n", num1, num2, product);
    printf("Division: %d / %d = %.2f\n", num1, num2, quotient);
    printf("Modulus: %d %% %d = %d\n", num1, num2, remainder);

    return 0;
}
```

2) Relational Operators

These are used to compare two quantities and depending on their relation take certain decision. The value of relational operator is either 1(if the condition is true) and 0 (if the condition is false).

Operator	Meaning	Example If a=20 and b=10
==	Equal to	(a==b) is not True
!=	Not Equal to	(a!=b) is True
>	Greater than	(a>b) is True
<	Less than	(a<b) is not True
>=	Greater than or Equal to	(a>=b) is True
<=	Less than or Equal to	(a<=b) is not True

3) Logical Operators

Logical Operators are used to compare or evaluate logical and relational expressions and returns either 0 or 1 depending upon whether expressions results true or false.

Operator	Meaning	Example If a=20 and b=10
&&	Logical AND	Expression ((a==b)&&(a>15)) equals to 0
||	Logical OR	Expression ((a==b)||(a>15)) equals to 1
!	Logical NOT	Expression !(a==b) equals to 1

4) Assignment Operators

Assignment Operators are used to assign the result of an expression to a variable. The mostly used assignment operator is "=".

Operator	Name	Example
=	Assignment	c=a+b will assign value of a+b into c
+=	Add AND assignment	c+=a is equivalent to c=c+a
-=	Subtract AND assignment	c-=a is equivalent to c=c-a
*=	Multiply AND assignment	c*=a is equivalent to c=c*a
/=	Divide AND assignment	c/=a is equivalent to c=c/a
%=	Modulus AND assignment	c%=a is equivalent to c=c%a

5) Increment and Decrement Operator

The increment operators (++) causes its operand to be increased by 1 whereas decrement operator (--) causes its operand to be decreased by 1.

They are unary operators (ie. They operate on single operand).It can be written as

- x++ or ++x (post and pre increment)
- x-- or - - x (post and pre decrement)

Note: The increment and decrement operators can be used as postfix and prefix notations.

Prefix notation: The variable is incremented (or decremented) first and then expression is evaluated using the new value of the variable.

5) Increment and Decrement Operator

Eg. int m=5;

y=++m;

In this case the value of y and m would be 6.

Postfix notation: The expression is evaluated first using the original value of the variable and then variable is incremented (or decremented by one).

Eg. int m=5;

y=m++;

The value of y would be 5 and m would be 6

5) Increment and Decrement Operator

Eg. int m=5;

y=++m;

In this case the value of y and m would be 6.

Postfix notation: The expression is evaluated first using the original value of the variable and then variable is incremented (or decremented by one).

Eg. int m=5;

y=m++;

The value of y would be 5 and m would be 6

6) Conditional Operator

The ternary Operator pair “?:” is available in c to construct conditional expressions of the form.

expression1? expression2: expression3

The expression1 is evaluated first,

- if it is true then the expression 2 is evaluated and its value becomes the value of the expression.
- If expression1 is false expression3 is evaluated and its value becomes the value of the expression.

6) Conditional Operator

The ternary Operator pair "?:" is available in c to construct conditional expressions of the form.

expression1? expression2: expression3

The expression1 is evaluated first,

- if it is true then the expression 2 is evaluated and its value becomes the value of the expression.
- If expression1 is false expression3 is evaluated and its value becomes the value of the expression.

Example:

Consider the following statements.

```
int a, b;
a=10;
b=15;
max= (a>b)? a: b;
```

In this example value of b will be assigned to max.

6) Conditional Operator

C Program to show conditional operator:

```
#include <stdio.h>

int main() {
    int a, b, max;
    a = 10;
    b = 15;
    max = (a > b) ? a : b;

    printf("Value of a: %d\n", a);
    printf("Value of b: %d\n", b);
    printf("The maximum value is: %d\n", max);
    return 0;
}
```

7) Bitwise operators

The operators which are used for the manipulation of data at bit level. These operators are used for testing the bits, shifting them right or left.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	Shift left
>>	Shift right

Operators: Bitwise Operators

The truth tables for &, |, and ^ is as follows

P	Q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Assume a = 60 and b = 13 in binary format, they will be as follows −

```
            a = 0011 1100
            b = 0000 1101
            -----------------
a&b     =   0000 1100 =>12
a|b     =   0011 1101 =>61
a^b     =   0011 0001 => 49
```

Bitwise shift operators

i. Left shift <<

This causes the operand to be shifted left by some bit positions.

General form:

Operand<<n

Eg.

n=60, n1=n<<3

N	0000 0000 0011 1100
First shift:	0000 0000 0111 1000
Second shift:	0000 0000 1111 0000
Third shift:	0000 0001 1110 0000

After left shifting value of n1 becomes 480

Bitwise shift operators

ii. Right shift >>

This causes operand to be shifted to the right by some bit positions.

Operand >>n

Eg. n=60, n2=n>>3

N	0000 0000 0011 1100
First shift	0000 0000 0001 1110
Second shift	0000 0000 0000 1111
Third shift	0000 0000 0000 0111

After Right shifting value of n2 becomes 7

8) Special operators

C supports some special operators. Some of them are

i. Sizeof operator

The sizeof is a compile time operator and when used with an operand, it returns the number of bytes the operand occupies. The operand may be variable, constant or data type qualifier.

Examples:

```
m=sizeof(sum);
n=sizeof(long int);
```

Operators:

Program

```
#include<stdio.h>
#include<conio.h>
int main()
{
float num;
printf("Number of bytes allocated =%d",sizeof(num));
getch();
return 0;
}
```

ii. Comma operator

The comma operator can be used to link related expressions together .A comma-linked list are evaluated left to right and the value of right most expression is the value of combined expression.

For example the statement

value = (x=10, y=5, x+y);

First assigns the value 10 to x then assigns 5 to y and finally assigns 15 (i.e. 10+5) to value.

Operator precedence and associativity

- Operator precedence is a predefined rule of priority of operators. If more than one operators are involved in an expression the operator of higher precedence is evaluated first.
- Associativity indicates the order in which multiple operators of same precedence executes.

Precedence	Operator	Associativity
1	(), {}	Left to right
2	++, --, !	Right to left
3	*, /, %	Left to right
4	+, -	Left to right
5	<, <=, >, >=	Left to right
6	== ,!=	Left to right
7	&&	Left to right
8	||	Left to right
9	?:	Right to left
10	=, *=, /=, %=, -=	Right to left

Some Questions to Work On:

- WAP to illustrate relational operators.
- WAP to illustrate assignment operators.
- WAP to illustrate logical operators.
- WAP to illustrate increment/decrement operators.
- WAP to illustrate bitwise operators.

Some Questions to Work On:

Q.1. Debug the following program:

```
#include <stdio.h>
int main() {
    int a, b;
    printf("Enter two numbers: ")
    scanf("%d %d", a, b);
    result = a + b;
    printf("The sum of %d and %d is %d\n", a, b result);
    return 0
}
```

Some Questions to Work On:

Q.1. Debug the following program. (Solution)

- Missing semicolon after the printf statement.
- scanf is missing the address-of operator (&).
- The variable result is used but not declared.
- Missing comma between format specifiers in printf.
- Missing semicolon at the end of return statement.

Some Questions to Work On:

Q.2. WAP to calculate area and circumference of circle having the radius r should be taken from user.

Q.3. WAP to swap two numbers.

Q.4. WAP to find the sum of digit of given (3 digit) number.

Q.5. WAP to find the reverse of a given (3 digit) number.

Q.2. WAP to calculate area and circumference of circle having the radius r should be taken from user.


```
#include<stdio.h>
#include<conio.h>
int main()
{
float r,area,circum;
printf("Enter the radious of circle");
scanf("%f",&r);
area=3.1416*r*r;
circum=2*3.1416*r;
printf("\nArea of circle=%f",area);
printf("\nCircumference of circle is %f",circum);
getch();
return 0;
}
```

Q.3. WAP to swap two numbers.


```
#include<stdio.h>
#include<conio.h>
int main()
{
int a,b,temp;
a=5;
b=10;
printf("Number before swapping a=%d and b=%d",a,b);
temp=a;
a=b;
b=temp;
printf("\nNumber after swapping a=%d and b=%d",a,b);
getch();
return 0;
}
```

Q.5. WAP to find the reverse of a given (3 digit) number


```
#include<stdio.h>
#include<conio.h>
int main()
{
int num,a,b,c,rev;
printf("Enter the number");
scanf("%d",&num);
c=num%10;
b=(num/10)%10;
a=num/100;
rev=c*100+b*10+a;
printf("Reverse of given number is %d",rev);
getch();
return 0;
}
```

Common Errors in Programming

Errors Vs Bug

Error

- An issue in the program that causes it to fail or behave unexpectedly.
- Detected by the compiler (syntax errors) or during runtime (runtime errors).
- Syntax errors, runtime errors, logical errors, semantic errors
- Fixed by correcting syntax or adding missing components.

Bug

- A flaw or fault in the code that leads to incorrect or unintended behavior.
- Refers specifically to mistakes in the code logic or implementation.
- A program calculating the wrong result due to incorrect logic is a bug.
- Fixed through debugging and modifying the faulty logic.

Common Errors in Programming

An **error** in programming refers to a mistake or issue in the code that prevents the program from compiling, running, or producing the correct output.

Syntax Errors

- Occur when code violates the language's grammar rules.
- Example: Missing semicolons, unmatched braces, incorrect keywords.
- **Fix**: Check for typos or missing punctuation.

  Example: int x = 5 // Missing semicolon

Compilation Errors

Arise during the compilation process due to invalid syntax or data types.

Fix: Review error messages and correct the code accordingly.

Logical Errors

The program runs but produces incorrect results due to faulty logic.

Example: Using the wrong operator in a calculation.

Fix: Test and debug the logic of the program thoroughly.

Common Errors in Programming

Runtime Errors

Occur while the program is running. Examples include dividing by zero, invalid memory access, and buffer overflow.

Fix: Handle exceptions properly and check for conditions that may cause such errors.

Example:

```
int x = 0;
printf("%d", 10 / x);  // Division by zero
```

Debugging Basis

Debugging Basis

- **Debugging** is the process of identifying, analyzing, and fixing errors or bugs in a program.
- It involves systematically testing the code, understanding the root cause of errors, and applying fixes to ensure the program works as intended.

Steps in Debugging:

- **Reproduce the Error**: Identify the exact input or condition that causes the error.
- **Analyze the Code**: Use tools like debuggers, print statements, or logs to trace the program flow.
- **Isolate the Bug**: Narrow down the part of the code where the issue occurs.
- **Fix the Bug**: Modify the code to correct the error.
- **Test the Fix**: Ensure that the program runs correctly after the fix and does not introduce new errors.

Debugging Basics

Debugging is usually done systematically:

- **Reproduce the Issue:** Begin by comprehending the issue. Continuously reproduce the issue to comprehend the issue's scale and impact.
- **Identify the Issue:** Determine the exact section of code where the problem occurs. Examining error message logs or utilizing a debugger is frequently required.
- **Determine the Root Cause:** Determine the root cause when you've isolated the problem. It could be a logical error, bad data, or an improper function call.
- **Resolve the Problem:** Create a solution to the bug. Check that the solution solves the root problem and does not bring new problems.
- **Test the correction:** Once the correction has been implemented, extensively test the code to ensure the issue has been handled. It could include executing unit, integration, or user tests.
- **Regression Testing:** Exercise caution while introducing new bugs and addressing existing ones. Regression testing should be performed to guarantee that existing functionality is not lost.
- **Documentation:** Write out the bug, how it was fixed, and any lessons learned. This helps in the exchange of information and the prevention of similar problems in the future.

Question Collection

1. What is data type? Why do we need it in programming? Explain any three basic data types with example.
2. What do you mean by unformatted I/O? Explain
3. Write short notes:
   - Operator precedence and associativity
   - History of C
   - Bitwise Operator
   - Local, Global, and Static variables
   - Conditional Operator
   - Pseudocode
   - Ternary/ conditional(:?) Operator
   - Comma operator
4. What are preprocessor directives? Discuss #define directive with example.
5. Discuss structure of a C Program with suitable example.
6. What is type conversion? Discuss type casting with suitable example.
7. Discuss increment and decrement operators with example.
8. What is algorithm? How is it different from flow chart?
9. What is variable? How is it different from constant? How do you write comments in C?
10. Explain formatted I/O functions in detail.
11. Differentiate between constant and literals. Why do we need to define the type of data?
12. How do you swap the values of two integers without using the third temporary variable? Justify with the example.
13. List different types of operators and explain any three of them.
14. WAP that will convert temperature in centigrade into Fahrenheit.
15. WAP to display your name.
16. WAP to calculate area and circumference of circle having the radius r should be taken from user.
17. WAP to find the reverse of a given (3 digit) number.
18. WAP in C to take two numbers as input from the user and perform basic arithmetic operations such as addition, subtraction, multiplication, division, and modulus

Also refer to class notes too.

THANK YOU

Any Queries ?