K. D. K. College of Engineering, Nagpur

Department of Mechanical Engineering

Subject: Programming for Problem Solving

Subject Code: 1BME04T

Semester: I

Subject Teacher: V. D. Dhopte, Assistant Professor

Unit-1: Introduction to C Programming

Communicating with a computer involves speaking the language the computer understands, which immediately rules out English as the language of communication with computer. However, there is language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs.

Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them – constants, variables and keywords are constructed, and finally how are these combined to form an instruction. A group of instructions would be combined later on to form a program. This is illustrated in the Figure 1.

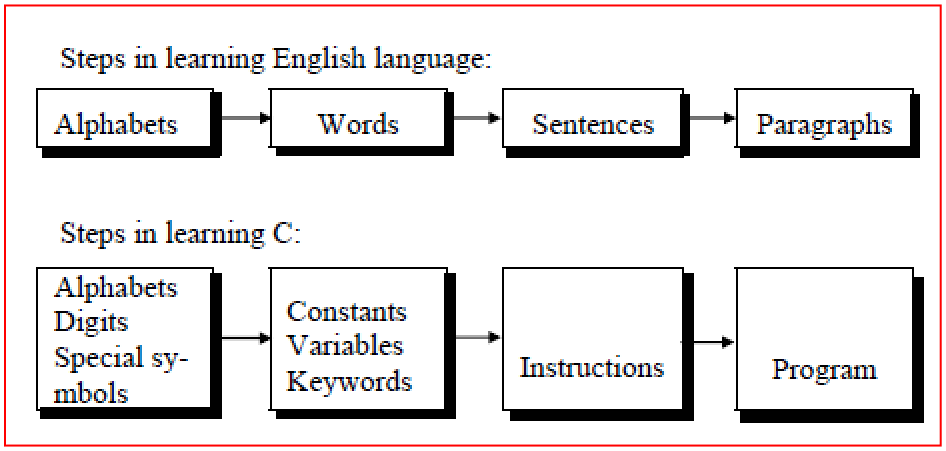

Steps in learning English language:
Alphabets
Words
Sentences
Paragraphs
Steps in learning C:
Alphabets Digits Special sy- mbols
Constants Variables Keywords
Instructions
Program

Figure 1

C is a programming language developed at AT& T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie.

<u>Basic Structure of C program</u>

The structure of a C program typically consists of several sections, some of which are optional. The only mandatory section is the main( ) function.

1. Documentation Section:

This section includes comments to enhance program readability and understanding. It can contain details like author, date, purpose, and any modifications. Comments in C are enclosed within /* ... */ for multi-line comments or // for single-line comments.

Example:

```
  /*
Program: Example C Program
Author: [Your Name]
Date: September 14, 2025
Description: This program demonstrates the basic structure of a C program.
*/
```

Or

```
/* Calculation of simple interest */
/* Author XYZ Date: 25/05/2004 */
```

Or

```
// This program demonstrates the basic structure of a C program.
```

2. Preprocessor Directives

These are lines included at the beginning of the program that provide instructions to the compiler. This section includes header files using the #include preprocessor directive. Header files contain declarations of standard library functions.

For example:

```
#include <stdio.h>// Includes the standard input/output library
```

This line tells the compiler to include the standard input-output header file, which is necessary for using functions like "printf" and "scanf".

3. The Main Function

Every C program must have a "main" function. This is the entry point where the execution of the program starts.

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

- int specifies that the main function will return an integer.
- return 0; indicates that the program executed successfully.

4. Variable Declarations

Before using variables, you must declare them.

For example:
```
int a;
float b;
```

5. Statements and Expressions

These are the instructions that define what the program does, such as assigning values to variables or performing operations.

```
a = 10;
b = 20.5;
```

6. Input/Output Functions

These functions are used to interact with the user for input or display results.

```
printf("Hello, World!\n");  // Output
scanf("%d", &a);            // Input
```

Example of a Simple C Program

Here's a complete example of a basic C program that prints "Hello, World!" to the console:

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

int main() {  // Main function
printf("Hello, World!\n");  // Print statement
return 0;  // Exit the program
}
```

Explanation:

1. **#include <stdio.h>**: Includes the standard I/O library for using printf.
2. **int main()**: Defines the main function where execution begins.
3. **printf("Hello, World!\n");**: Prints the text to the console.
4. **return 0;**: Ends the “main” function and returns 0 to the operating system.

Features of C Language

1. Simple and Efficient

- **Minimalistic Design**: C has a straightforward syntax, which makes it relatively easy to learn and use. Its simplicity also contributes to efficient execution and compilation.
- **Low-Level Access**: It provides direct access to hardware and memory via pointers, which allows for fine-tuned optimization and efficient resource management.

2. Structured Programming

- **Modular Code**: C supports functions and procedures, which help in breaking down complex problems into smaller, manageable units. This promotes code reuse and modularity.
- **Control Structures**: C includes control structures like if, else, while, for, and switch that enable structured and logical flow of control.

3. Portability

- **Cross-Platform**: C code can be compiled and run on different types of systems with minimal modification. This is facilitated by its standardization in the ANSI C standard (also known as C89 or C90).

4. Rich Library Support

- **Standard Library**: The C Standard Library provides a comprehensive set of built-in functions for tasks such as input/output, string manipulation, mathematical computations, and memory management.

5. Efficiency

- **Performance**: C is known for its high performance and efficiency. The language's ability to interact closely with hardware, combined with its efficient compilation, results in programs that are fast and resource-conserving.

6. Pointer Arithmetic

- **Direct Memory Access**: C supports pointers, which allow direct manipulation of memory addresses. This feature enables efficient array handling, dynamic memory management, and system-level programming.

7. Dynamic Memory Management

- **Memory Allocation**: C provides functions like malloc, calloc, realloc, and free for dynamic memory allocation and deallocation, giving programmers control over memory usage.

8. Recursion

- **Function Calling**: C supports recursive functions, where a function can call itself. This is useful for solving problems that can be broken down into simpler, similar problems.

9. Preprocessor Directives

- **Code Management**: The C preprocessor allows for conditional compilation, macro definitions, and file inclusion. This can simplify code management and increase flexibility.

10. Extensibility

- **Custom Data Types**: C allows the creation of user-defined data types through structures (struct), unions (union), and enumerations (enum), enhancing the ability to model complex data.

11. Standardization

- **C Standards**: Over the years, C has evolved through several standards, including ANSI C (C89/C90), C99, C11, and C18, which have introduced various enhancements and features while maintaining backward compatibility.

12. Compatibility

- **Interoperability**: C is compatible with other languages and can be used to interface with assembly language. It is often used for systems programming, embedded systems, and in conjunction with other languages for high-performance applications.

Summary

C’s combination of simplicity, efficiency, and control makes it a powerful tool for a variety of applications, from system programming and embedded systems to application development. Its features have laid the groundwork for many modern programming languages and paradigms.

Character Set

In C programming, the character set refers to the set of characters that are valid for use in a C program. This includes letters, digits, and various symbols. Here's a breakdown of the components of the C character set:

1. Letters

- **Uppercase Letters**: A to Z (ASCII values 65 to 90)
- **Lowercase Letters**: a to z (ASCII values 97 to 122)

2. Digits

- **Numeric Digits**: 0 to 9 (ASCII values 48 to 57)

3. Special Characters

- **Punctuation Marks and Symbols**: Includes characters such as !, @, #, $, %, ^, &, *, (,), -, _, =, +, {, }, [,], |, \, ;, :, ', ", <, >, ,, ., ?, /, and ~.

4. Whitespace Characters

- **Space**: The space character (ASCII value 32)
- **Tab**: Horizontal tab (ASCII value 9)
- **Newline**: Line feed (ASCII value 10)
- **Carriage Return**: (ASCII value 13)

5. Escape Sequences

- **Escape Sequences**: Special characters represented by a backslash (\) followed by a specific character:
  - \n (newline)
  - \t (tab)
  - \\ (backslash)
  - \" (double quote)
  - \' (single quote)
  - \r (carriage return)
  - \b (backspace)
  - \f (form feed)

6. Comments

- **Single-Line Comments**: // This is a comment
- **Multi-Line Comments**: /* This is a comment */

7. Unicode

- **Character Representation**: In standard C (especially before C99), characters are typically represented using the ASCII character set. However, with the introduction of C99 and later standards, support for wide characters and Unicode via wchar_t and related functions has been introduced.

Summary

The C character set encompasses all characters that can be used in a C program, including letters, digits, special characters, and whitespace. This set is fundamental for writing source code, as it defines the symbols that can be used for identifiers, literals, operators, and other elements of the language.

C Tokens:

In C programming, the smallest individual units are known as c tokens. C has six types of tokens as shown in figure 2. They are the building blocks of C programs. C programs are written using these tokens and the syntax of the language.

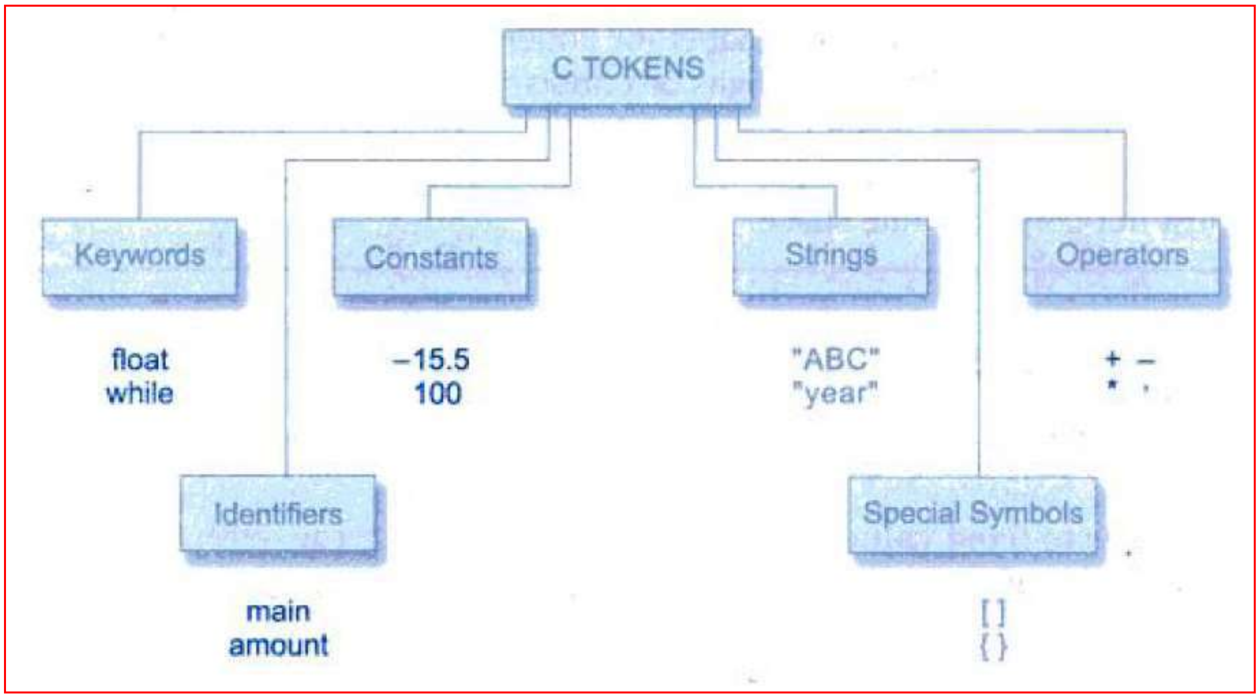
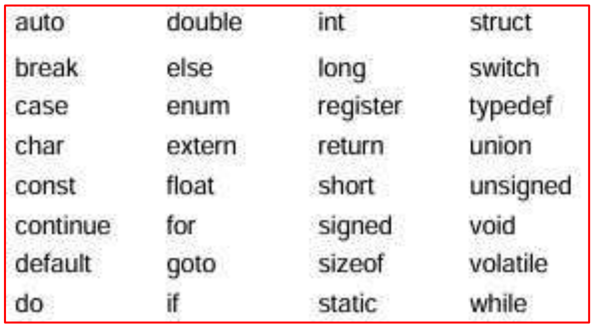


Figure 2: C tokens and examples

C Keywords

Keywords are the words whose meaning has already been explained to the C compiler (or in a broad sense to the computer). The keywords cannot be used as variable names. If it is used, then new meaning will be assigned to the keyword, which is not allowed by the computer. Some C compilers allow you to construct variable names that exactly resemble the keywords. However, it would be safer not to mix up the variable names and the keywords. The keywords are also called 'Reserved words'.

There are only 32 keywords available in C. Figure 3 gives a list of these keywords for your ready reference.



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

Figure 3: Keywords

Identifiers

Identifiers refer to the names of variables, functions and arrays. These are user-defined names.

Rules for Identifiers:

1. First character must be a letter (a-z, A-Z) or an underscore (_), followed by letters, digits (0-9), or underscores.

2. Must consist of only letters, digits or underscore.

3. Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used.

4. Cannot use a keyword.

5. Must not contain white space.

Examples: sum, total_amount, initial_0, num_5_sum

Constant

A constant is an entity that doesn't change.

Types of C Constants C constants can be divided into two major categories:

(a) Primary Constants

(b) Secondary Constants

These constants are further categorized as shown in Figure 4.

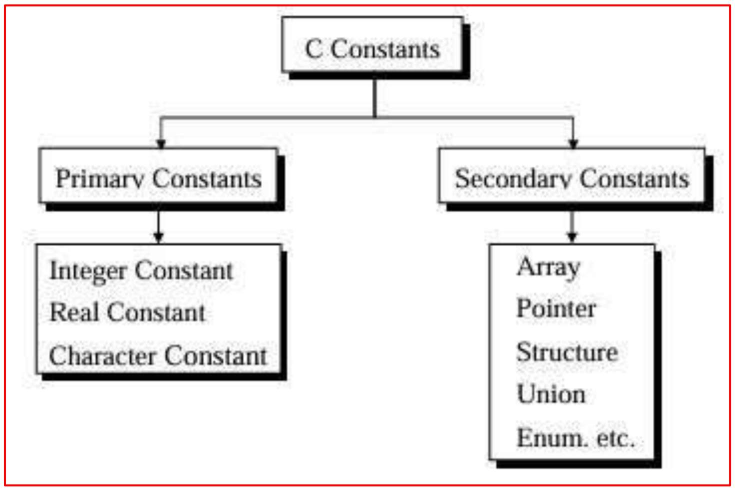


Figure 4

Rules for Constructing Integer Constants

(a) An integer constant must have at least one digit.

(b) It must not have a decimal point.

(c) It can be either positive or negative.

(d) If no sign precedes an integer constant it is assumed to be positive.

(e) No commas or blanks are allowed within an integer constant.

(f) The allowable range for integer constants for a 16-bit compiler is -32768 to 32767.

Example: 426, +26, -500,

Rules for Constructing Real Constants

Real constants are often called Floating Point constants. The real constants could be written in two forms—Fractional form and Exponential form.

Following rules must be observed while constructing real constants expressed in fractional form:

(a) A real constant must have at least one digit.

(b) It must have a decimal point.

(c) It could be either positive or negative.

(d) Default sign is positive.

(e) No commas or blanks are allowed within a real constant.

Example: 325.34, +105, -52.75

The exponential form of representation of real constants is usually used if the value of the constant is either too small or too large. It however doesn’t restrict us in any way from using exponential form of representation for other real constants.

In exponential form of representation, the real constant is represented in two parts. The part appearing before ‘e’ is called mantissa, whereas the part following ‘e’ is called exponent. Following rules must be observed while constructing real constants expressed in exponential form:

(a) The mantissa part and the exponential part should be separated by a letter e.

(b) The mantissa part may have a positive or negative sign.

(c) Default sign of mantissa part is positive.

(d) The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.

(e) Range of real constants expressed in exponential form is -3.4e38 to 3.4e38.

Example: +3.2e-5, 4.1e8, -0.2e+3, -3.2e-5

Rules for Constructing Character Constants

(a) A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas. Both the inverted commas should point to the left. For example, ’A’ is a valid character constant whereas ‘A’ is not.

(b) The maximum length of a character constant can be 1 character.

Example: ’A’, ’5’, ’=’

Variables

A variable is an entity that may change. In any program calculations are performed. The results of these calculations are stored in computers memory. The calculated values are stored in these memory. Since the value stored in each location may change the names given to these locations are called variable names.

Types of C Variables

An entity that may vary during program execution is called a variable. Variable names are names given to locations in memory. These locations can contain integer, real or character constants.

A particular type of variable can hold only the same type of constant. For example, an integer variable can hold only an integer constant, a real variable can hold only a real constant and a character variable can hold only a character constant.

The rules for constructing different types of constants are different. However, for constructing variable names of all types the same set of rules apply. These rules are given below.

Rules for Constructing Variable Names

(a) A variable name is any combination of 1 to 31 alphabets, digits or underscores. Some compilers allow variable names whose length could be up to 247 characters. Still, it would be safer to stick to the rule of 31 characters. Do not create unnecessarily long variable names as it adds to your typing effort.

(b) The first character in the variable name must be an alphabet or underscore.

(c) No commas or blanks are allowed within a variable name.

(d) No special symbol other than an underscore (as in gross_sal) can be used in a variable name.

Example: si_int, m_hra, pop_e_89

Data Types

Data types specify the type of data a variable can hold. Each data type determines the size and type of data that can be stored and how the data is interpreted. C language is rich in its data types. The variety of data types available, allow the programmer to select the type, appropriate to the needs of the application as well as the machine.

There are three classes of data types:

1. Primary (or fundamental) data types

The primary data types are

a) Integer Type: Represent whole number – short int, int, long int

b) Floating Point Type: Represent real numbers (numbers with decimal points) – float, double, long double

c) Character Types: Represents single characters – char

2. Derived data types

The derived data types are arrays, functions, structures and pointers.

3. User-defined data types

Details of some data types are given in below table on a 16-bit machine.

Sr. No.	Data Type	Keyword	Size (bits)	Range
1	Short Integer	short int short	8	-128 to 127
2	Integer	int	16	-32,768 to 32,767
3	Long Integer	long int or long	32	-2,147,483,648 to 2,147,483,647
4	Floating Point	float	32	3.4E-38 to 3.4E+38
5	Double-precision floating point	Double	64	1.7E-308 to 1.7E+308
6	Extended double-precision floating point	Long double	80	3.4E-4932 to 1.1E+4932
7	Character	char	8	-128 to 127

Example:

```
#include <stdio.h>

int main()
{
        int age = 30;                // Integer type
        float height = 5.9;          // Floating-point type
        char initial = 'A';          // Character type

        printf("Age: %d\n", age);
        printf("Height: %.1f\n", height);
        printf("Initial: %c\n", initial);

        return 0;
}
```

Unit-2

Operators and Expressions

Introduction

C supports a rich set of built-in operators. An operator is symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.

C operators can be classified into a number of categories. They include:

1) Arithmetic operators
2) Relational operators
3) Logical operators
4) Assignment operators
5) Increment and decrement operators
6) Conditional operators
7) Bitwise operators

1) Arithmetic operators

C provides all the basic arithmetic operators. These operators perform basic mathematical operations:

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulo Division)

Note:

- The modulo division operator % cannot be used on floating point data.
- C does not have an operator for exponentiation.
- Variables in arithmetic expression are known as operands.

Integer Arithmetic
When the operands in a single arithmetic expression such as a+b are integers, the expression is called an integer expression and the operation is called integer arithmetic. Integer arithmetic always yields and integer value.

Example:
```
int a = 10, b = 3;
int sum = a+b;              // 13
int difference = a-b;       // 7
int product = a*b;          // 30
int quotient = a/b;         // 3
int remainder = a%b;        // 1
```

Real Arithmetic
An arithmetic operation involving only real operands is called real arithmetic.

Example:
```
float x=6.0, y=7.0, z;
z=x/y;              //0.857143
```

Mixed-mode Arithmetic
When one of the operands is real and the other is integer, the expression is called mixed-mode arithmetic expression. If anyone operand is of the real type, then the real operation is performed and the result is always a real number.
Example:
```
int x=15;
float y=10.0;
```

x/y = 15/10.0 = 1.5

2) Relational operators

These operators are used to compare two values to take certain decisions. An expression containing a relational operator is termed as relational expression. The relational operators are as below:

- == (Equal to)
- != (Not equal to)
- > (Greater than)
- < (Less than)
- >= (Greater than or equal to)
- <= (Less than or equal to)

Example:

```
if(a > b)
{
// Do something
}
```

Relational expressions are used in decision statements such as “if” and “while” to decide the course of action of a running program. When arithmetic expressions are used on either side of a relational operator such as “a+b>c+d”, the arithmetic expressions will be evaluated first and then the results compared.

3) Logical operators

C has the following three logical operators.

- && (Logical AND)
- || (Logical OR)
- ! (Logical NOT)

The logical operators are used when there is a requirement to test more than one condition and make decisions. An example is: a>b && x==10. An expression of this kind, which combines two or more relational expression, is termed as a logical expression.

Example:

```
if (a >5&& b <5)
{
// Do something
}
```

4) Assignment operators

Assignment operators are used to assign the result of an expression to a variable. C has set of shorthand assignment operators as given below,

- = (Assignment)
- += (Add and assign)
- -= (Subtract and assign)
- *= (Multiply and assign)
- /= (Divide and assign)
- %= (Modulus and assign)

Example:

```
a += b;    // Equivalent to a = a + b;
a -= b     // Equivalent to a = a – b;
a *= b     // Equivalent to a = a * b;
a /= b     // Equivalent to a = a / b;
a %= b     // Equivalent to a = a % b;
```

Consider, if initial value of a=2 and b=3
a +=b   //a=5
a = a+b = 2+3= 5
Then new value of a will be 5.

a -=b   //a= -1
a = a-b = 2 – 3 = -1

5) Increment and Decrement operators

I) Increment Operator (++)

- **Prefix Increment (++variable)**: Increases the value of the variable by 1, then returns the updated value.

  ```
  a = 5;
  b = ++a;          //a = 6, b = 6
  ```

- **Postfix Increment (variable++)**: Returns the current value of the variable, then increases the value by 1.

  ```
  a = 5;
  c = a++;          //c = 5, a = 6
  ```

Examples:

```
#include <stdio.h>
#include<conio.h>
void main()
{
inta,b,c;
    a=5;
    // Prefix increment
   b = ++a;              // a becomes 6, b is 6
printf("After prefix increment: a = %d, b = %d", a, b);

    // Postfix increment
    c = a++; // c is 6, a becomes 7
printf("After postfix increment: a = %d, c = %d", a, c);
getch();
}
```

II) Decrement Operator (--)

- **Prefix Decrement (--variable)**: Decreases the value of the variable by 1, then returns the updated value.

  ```
  a = 5;
  b = --a;        //a = 4, b = 4
  ```

- **Postfix Decrement (variable--)**: Returns the current value of the variable, then decreases the value by 1.

  ```
  a = 5;
  c = a--;        //c = 5, a = 4
  ```

Examples:

```
#include <stdio.h>
#include<conio.h>
void main()
{
inta,b,c;
a = 5;
    // Prefix decrement
b = --a;   // a becomes 4, b is 4
printf("After prefix decrement: a = %d, b = %d", a, b);

    // Postfix decrement
c = a--;   // c is 4, a becomes 3
printf("After postfix decrement: a = %d, c = %d", a, c);
getch();
}
```

6) Conditional operators

A ternary operator pair “? :” is available in C to construct conditional expressions of the form

variable = expression 1 ? expression 2 : expression3

The operator “? :” works as follows:

Expression1 is evaluated first,

- If expression1 is TRUE then the expression2 is evaluated and becomes the value of variable
- If expression1 is FALSE then the expression3 is evaluated and becomes the value of variable

Example:

```
a = 10;
b = 15;
x = (a > b) ?a : b;          // x = 15
```

7) Arithmetic Expressions

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. C can handle any complex mathematical expressions. C does not have an operator for exponentiation.
The examples of C expressions are given in below table.

Sr. No.	Algebraic Expression	C Expression
1.	$a \times b - c$	a*b–c
2.	$(m + n)(x + y)$	(m+n)*(x+y)
3.	$\left(\frac{ab}{c}\right)$	a*b/c
4.	$3x^2 + 2x + 1$	3*x*x+2*x+1
5.	$\frac{x}{y} + c$	x/y+c

8) Evaluation of Expression

Expressions are evaluated using an assignment statement of the form;

$$vaiable = expression;$$

- Variable is any valid C variable name.
- When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left hand side.
- All variables used in the expression must be defined before they are used in the expression.
- All variables used in the expression must assigned values before carrying out evaluation.

Examples of evaluation statements are

x = a * b – c;

y = b / c + d;

9) Precedence of Arithmetic Operators

- An arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators.
- The arithmetic operators : *, /, % have high priority.
- The arithmetic operators: +, – have low priority.
- The basic evaluation procedure includes two left-to-right passes through the expression. During first pass, the high priority operators are applied as they encountered. During second pass, the low priority operators are applied as they encountered.

  Example: x = a – b/3;

  consider a = 9, b = 12
  then, x = 9 – 12/3;
  the evaluation will be as given below,
  x = 9 – 4;
  x = 5;
- Whenever parentheses are used, the expressions within parentheses assume highest priority.

  Example: x = (a+c) – b/3;

  consider a = 9, b = 12, c = 2
  then, x = (9+2) – 12/3;
  the evaluation will be as given below,
  x = 11 – 12/3;
  x = 11 – 4;
  x = 7;

10) Type Conversion in Expressions

10.1) Implicit Type Conversion

C permits mixing of constants and variables of different data types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without loosing any significance. This automatic conversion is known as implicit type conversion.

If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds i.eC automatically converts smaller data types to larger data types when performing operations.

- **Integer Promotion**: `char` and `short` types are promoted to `int` when evaluated in an expression.
- **Mixed Type Operations**: If an operation involves both integers and floating-point types, the integer is converted to a floating-point type.

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it.

Example:

```
#include <stdio.h>
#include <conio.h>
void main()
{
char c = 'A';       // ASCII value 65
int a = 10;
float b = 5.5, result1, result2;

// Implicit conversion: c is promoted to int
        result1 = c + a;       // c is treated as 65 + 10 = 75.0
        printf("Result1: %.2f\n", result1);

// Implicit conversion: i is promoted to float
        result2 = a + b;       // 10 + 5.5 = 15.5
        printf("Result2: %.2f\n", result2);

getch();
}
```

10.2) Explicit Conversion

Sometimes the user wants to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of females to males in a town.

```
        ratio = female_number / male_number
```

Consider, female_number and male_number are declared as integers in the program, the decimal part of the result of the division would be lost and ratio would represent a wrong value. This problem can be solved by converting locally one of the variables to the floating point as shown below:

```
        ratio = (float) female_number / male_number
```

The operator (float) converts the female_number to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus keeping the fractional part of result.

Note that the operator (float) affect the value of the variable female_number. In other parts of the program the type of female_number remains as int.

The process of such local conversion is known as explicit conversion or casting a value.

The general form is:

```
        (type-name) expression
```

where type-name is one of the standard C data types and the expression may be a constant, variable or an expression.

Example:

x = (int) 7.5	result would be 7
x = (float) 10/4	result would be 2.5
x = (int)21.3/(int)4.5	Evaluated as 21/4 and the result would be 5

11) Operator precedence and associativity

In C programming each operator has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. The different operators have distinct levels of precedence. The operators at the higher level of precedence are evaluated first. This is known as the associativity property of an operator.

The below table shows list of operators, their precedence levels, and rules of association. Rank1 indicates the highest precedence level. It is very important to note carefully, the order of precedence and associativity of operators.

Operator	**Description**	**Associativity**	**Rank**
() []	Function Call Array element reference	Left to Right	1
* / %	Multiplication Division Modulus	Left to Right	3
+ –	Addition Subtraction	Left to Right	4
< <= > >=	Less than Less than equal to Greater than Greater than equal to	Left to Right	6
== !=	Equality Inequality	Left to Right	7
&&	Logical AND	Left to Right	11
| |	Logical OR	Left to Right	12

Consider the following conditional statement

If (x == 10 + 15 && y < 10)

The precedence rule say that the addition operator has higher priority than the logical operator (&&) and the relational operators (== and <). The above statement will become

If (x == 25 && y < 10)

The next step is to determine whether x is equal to 25 and y is less than 10.

Assume, x = 20 and y = 5

Then, x == 25 is FALSE (0)

y< 10 is true (1)

Since the operator '<' has higher priority compared to ==, y < 10 is tested first and then x == 25 is tested.

Finally: if (FALSE && TRUE)

12) Bitwise Operators

These operators perform bit-level operations:

- `&` (Bitwise AND)
- `|` (Bitwise OR)
- `^` (Bitwise XOR)
- `~` (Bitwise NOT)
- `<<` (Left shift)
- `>>` (Right shift)

Example:

```
int c = a & b; // Bitwise AND
```

Unit-III

Decision Making with 'if' statement

In practice, the order of execution of statement needs to be change based on certain conditions. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.

- The 'if' statement is a powerful decision-making statement.
- It is used to control the flow of execution of statements.
- It is basically a two-way decision statement. It takes the following form

  ```
  if (test expression)
  ```
- It allows the computer to evaluate the expression first and then depending upon whether the value of the expression is 'true' or 'false', it transfer the control to a particular statement. This point of program has two paths to follow, one for 'true' condition and the other for 'false' condition as shown in below figure.

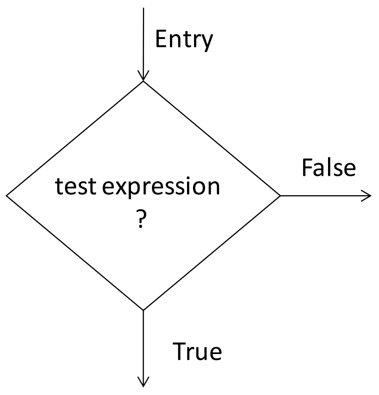


Fig.: Two-way branching

Example:

```
if(room is dark)
{
put on lights
}
```

The 'if' statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are

a) simple 'if' statement

b) 'if…else' statement

c) Nested 'if….else' statement

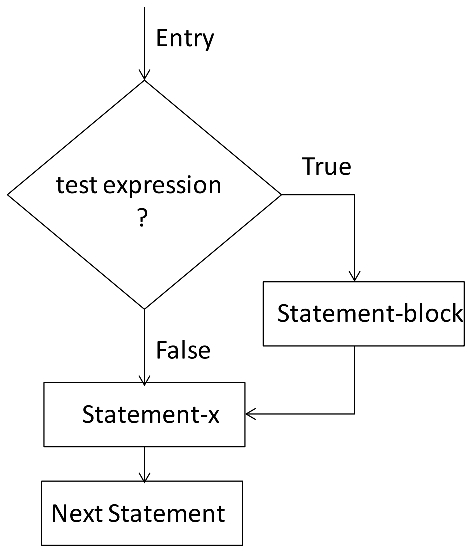
d) 'else if' ladder

Simple 'if' Statement

The general form of 'if' statement is

```
        if(test expression)
        {
        Statement-block;
        }
        statement-x;
```

If the 'test expression' is true, the statement-block will be executed; otherwise the statement-block will be skipped and the execution will jump to the statement-x.
When the condition is true both the statement-block and the statement-x are executed. It is shown in below figure.



Example:

```
if(category == sports)
{
        marks = marks + bonus_marks;
}
printf("%d", marks);
```

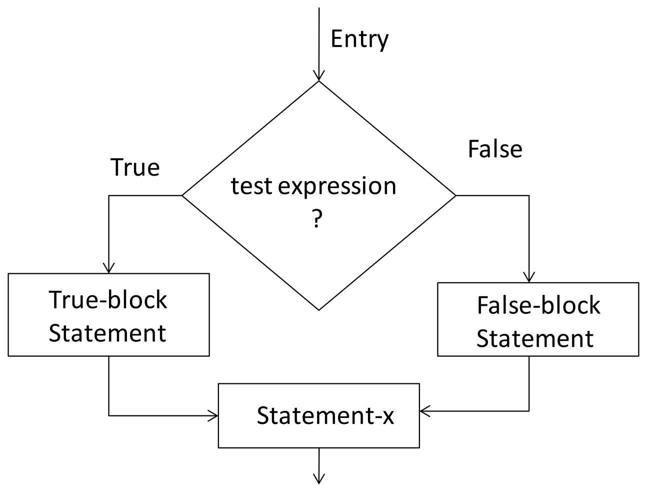
If the student belongs to the sports category, then bonus_marks are added to marks before they are printed. For others bonus_marks are not added.

The ‘if…else’ Statement

The ‘if…else’ statement is an extension of the simple ‘if’ statement. The general form is

```
        if (test expression)
        {
                True –block statement;
        }
        else
        {
                False –block statement;
        }
        statement-x
```

If the ‘test expression’ is ‘true’ then the true –block statement immediately following ‘if’ statement is executed.
If the ‘test expression’ is ‘false’ then the false –block statement immediately following ‘else’ statement is executed.
Either true-block or false-block will be executed at a time, not both.
It is shown in below figure.



Nesting of ‘if….else’ Statement

When a series of decisions are involved, user may have to use more than one ‘if….else’ statement in nested form as shown below.

```
if (test condition-1)
{
        if (test condition-2)
        {
                statement-1;
        }
        else
        {
                statement-2;
        }
}
else
{
        statement-3;
}
statement-x;
```

If test condition-1 is true, then test condition-2 will be checked. If test condition-2 is true, then statement-1 will be executed. If test condition-2 is false, then statement-2 will be executed. At last statement-x will be executed.

If test condition-1 is false, then statement-3 will be executed. At last statement-x will be executed.

The ‘Switch’ Statement

- C has a built-in multiway decision statement known as a ‘switch’.
- The general form of the ‘switch’ statement is as shown below:

```
switch (variable or expression)
{
        case value-1:
                block-1;
                break;
        case value-2:
                block-2;
                break;
        default:
                default-block;
                break;
}
statement-x;
```

- The ‘switch’ statement tests the value of a given variable or expression against a list of case values value-1, value-2 and so on, if a case is found whose value matches with the value of the expression, then the block of statements associated with that case is executed.
- The break statement at the end of each block signals the end of a particular case and causes an exit from the ‘switch’ statement.
- The default is an optional case. When it present, it will be executed if the value of the expression does not match with any of the case values.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int marks;
clrscr();
printf(“Enter the marks”);
scanf(“%d”, &marks);
index = marks/10;
switch (index)
{
    case 10:
```

```
    case 9:
    case 8:
            printf(“Honours”);
            break;
    case 7:
    case 6:
            printf(“First Division”);
            break;
    case 5:
            printf( “Second Division”);
            break;
    case 4:
            printf(“Third Division”);
            break;
    default:
            printf(“Fail”);
            break;
}
getch();
}
```

The ‘while’ Statement

The simplest of all the looping structures in C is the ‘while’ statement. The general form of the ‘while’ statement is given below.

```
while (test condition)
{
        body of the loop
}
```

- The test-condition is evaluated and if the condition is true, then the body of the loop is executed.
- After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again.
- This process of repeated execution of the body continues until the test-condition finally becomes ‘false’ and exit the loop.
- On exit, the program continues with the statement immediately after the body of the loop.

Example:

```
sum = 0;
n = 1;
while (n <= 10)
{
        sum = sum + n;
        n = n + 1;
}
printf(“%d”, sum);
```

The body of loop is executed 10 times for n = 1, 2, …., 10. For n = 11, condition becomes false and loop is exited. The variable ‘n’ is called control variable.

The ‘for’ Statement

The ‘for’ loop provides a more concise loop control structure. The general form of the ‘for’ loop is

```
    for ( initialization ; test-condition ; increment )
    {
        body of the loop
    }
```

The execution of the ‘for’ statement is as follows:

a) Initialization of the control variables is done first, using assignment statements such as i=1.

b) The value of the control variable is tested using the test-condition. The test-condition is relational expression such as i<10 that determines when the loop will exit. If the condition is true, the body of the loop is executed, otherwise the loop is terminated.

c) The control variable is incremented every time using an assignment statement such as i=i+1.

Example:

```
for ( i = 1; i <=10; i=i+1)
{
    printf(“%d”, i);
}
```

The ‘for’ loop is executed 10 times and prints the digits 1 to 10.

1) Write a program which reads three-digit integer no. from the user and checks weather the entered no. is Armstrong no. or not and displays an appropriate message on user screen.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num,ori,d1,d2,sum;
clrscr();
printf("Enter 3 digit integer number");
scanf("%d", &num);
ori = num;
d1 = num%10;
num = num/10;
d2 = num%10;
num = num/10;
sum = (d1*d1*d1) + (d2*d2*d2) + (num*num*num);
if(sum == ori)
{
printf("Number is an Armstrong number");
}
else
{
printf("Number is not an Armstrong number");
}
getch();
}
```

2) Write a program which reads three-digit integer no. from the user and checks weather the entered no. is Palindrome no. or not and displays an appropriate message on user screen.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num,ori,d1,d2,rev;
clrscr();
printf("Enter 3 digit integer number");
scanf("%d", &num);
ori = num;
d1 = num%10;
num = num/10;
d2 = num%10;
num = num/10;
rev = (d1*100) + (d2*10) + num;
if(rev == ori)
{
printf("Number is Palindrome number");
}
else
{
printf("Number is not Palindrome number");
}
getch();
}
```

3) Write a program to check if an integer number entered by the user is even or odd.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num;
clrscr();
printf("Enter the integer number");
scanf("%d", &num);
if(num % 2 == 0)
{
printf("Number is Even");
}
else
{
printf("Number is Odd");
}
getch();
}
```

4) Write a program which reads an integer number from the user and checks weather the entered number is positive, negative or zero. Also display an appropriate message on the screen.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num;
clrscr();
printf(“Enter the integer number”);
scanf(“%d”, &num);
if(num>0)
{
printf(“Number is positive”);
}
if(num<0)
{
printf(“Number is negative”);
}
if(num == 0)
{
printf(“Number is zero”);
}
getch();
}
```

5) Write a program which reads three integer numbers from the user and display the largest number on the user screen.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c,max;
clrscr();
printf(“Enter 3 integer number”);
scanf(“%d%d%d”, &a,&b,&c);
max = a;
if(b > max)
{
max = b;
}
if(c > max)
{
max = c;
}
printf(“Largest Number = %d”, max);
getch();
}
```

6) Write a program which reads X and Y co-ordinates from the user and checks whether the points are collinear or not and displays an appropriate message on user screen.

```
#include<stdio.h>
#include<conio.h>
void main()
{
float x1,y1,x2,y2,x3,y3,m1,m2;
clrscr();
printf("Enter x & y coordinates of 3 points");
scanf("%f%f%f%f%f%f", &x1,&y1, &x2,&y2, &x3,&y3);
m1 = (y2 – y1) / (x2 – x1);
m2 = (y3 – y1) / (x3 – x1);
if(m1 == m2)
{
printf("Points are collinear");
}
else
{
printf("Points are not collinear");
}
getch();
}
```

7) Write a program which reads X and Y coordinates of a point from the user and checks weather the entered point is in I,II,III,IV quadrant, point on X-axis, Y-axis or point is on the origin and displays the appropriate message on the user screen.

```
#include<stdio.h>
#include<conio.h>
void main()
{
float x,y;
clrscr();
printf("Enter x & y coordinates of point");
scanf("%f%f", &x,&y);
if(x > 0 && y > 0)
{
printf("Point lies in the I quadrant");
}
if(x < 0 && y > 0)
{
printf("Point lies in the II quadrant");
}
if(x < 0 && y < 0)
{
printf("Point lies in the III quadrant");
}
if(x > 0 && y < 0)
{
printf("Point lies in the IV quadrant");
}
if(x != 0 && y == 0)
{
printf("Point lies on the X-axis");
}
if(x == 0 && y != 0)
{
printf("Point lies on the Y-axis");
}
if(x == 0 && y == 0)
{
printf("Point lies on the Origin");
}
getch();
}
```

Unit-4: Arrays

In many applications, one needs to handle a large volume of data in terms of reading, processing and printing. To process such large amount of data one needs a powerful data type that would facilitate efficient storing, accessing and manipulation of data items. C supports a derived data type known as array that can be used for such applications. In its simplest form, an array can be used to represent a list of numbers or list of names. For example test score of a class of students.

One Dimensional Arrays

List of items can be given one variable name using only one subscript and such a variable is called a one-dimensional array. In mathematics, one often deals with variables that are single subscripted. For example

$$averagemarks = \frac{\sum_i^n X_i}{n}$$

The above equation is use to calculate the average of n values of x.In C, single-subscripted variable X_i can be expressed as,

X[1], X[2], X[3],----,X[n]

For example, if user want to represent a set of five numbers, say (35, 40, 20, 57, 19) by an variable number, then it is declared as below,

```
int number[5];
```

The computer reserves five storage locations.

Declaration of One Dimensional Arrays

Like other variables, arrays must be declared before they are used so that the compiler can allocate space for them in memory. The general form of array declaration is

```
type variable-name [size];
```

The type specifies the type of element that will be stored in the array, such as int, float, or char and the size indicates the maximum number of elements that can be stored inside the array.
For example

```
float height[50];
```

It declares the height to be an array containing 50 real elements.

Similarly, `int group[10];`
It declares the group as an array to contain 10 integer elements.

Initialization of One Dimensional Arrays

After an array is declared, its elements must be initialized. Otherwise they will contain garbage. An array can be initialized at either of the following stages

- At compile time
- At run time

Compile Time Initialization

One can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is

```
type array-name[size] ={List of values};
```

The values in the list are separated by commas. For example

```
int number[3] = {20, 10, 40};
```

It will declare the variable number as an array of size 3 and will assign values 20, 10, 40 to each element.

Run Time Initialization

An array can be initialized at run time. This approach is usually applied for initializing large arrays. For example

```
int marks[65];
for{i=1; i<=65; i=i+1}
{
        printf(“Enter marks of students”);
        scanf(“%d”, &marks[65]);
}
```

1) Write a program to find average marks obtained by a class of 65 students in a test.

```
/*Calculation of average marks*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        inti,sum=0,average;
        int marks[65];                          /*array declaration*/
        clrscr( );
        for(i=1; i<=65; i=i+1)
        {
                printf("Enter marks");
                scanf("%d", &marks[i]);         /*store data in array*/
        }
        for(i=1; i<=65; i=i+1)
        {
                sum = sum + marks[i];           /*read data from an array*/
        }
        average = sum/65;
        printf("%d", average);
        getch( );
}
```

2) Write a program which reads an integer array of dimension 50 from the user, reads any number from the user and counts the occurrence of the user entered number in the array and display the value of count on the user screen.

```
/*To count occurrence of user entered number*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        inta[50], i,num, count=0;
        clrscr( );
        for(i=1; i<=50; i=i+1)
        {
                printf("Enter the elements");
                scanf("%d", &a[i]);
        }
        printf("Enter the number");
        scanf("%d", &num);
        for(i=1; i<=50; i=i+1)
        {
                if(a[i] = = num)
                {
                count = count + 1;
                }
        }
        printf("%d", count);
        getch( );
}
```

3) Write a program which reads 70 integer numbers from the user, count the number of positive, negative and zero elements stored in the array and displays it on the user screen.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a[70], i, pcount=0, ncount=0, zcount=0;
        clrscr( );
        for (i=1; i<=70; i=i+1)
        {
                printf("Enter the element");
                scanf("%d", &a[i]);
        }
        for (i=1; i<=70; i=i+1)
        {
                if (a[i] > 0)
                {
                        pcount = pcount + 1;
                }
                if (a[i] < 0)
                {
                        ncount = ncount + 1;
                }
                if (a[i] == 0)
                {
                        zcount = zcount + 1;
                }
        }
        printf("%d%d%d", pcount, ncount, zcount);
        getch( );
}
```

Characteristics of One-Dimensional Arrays:

- **Fixed size**: The size of the array is defined at the time of creation and cannot be changed.
- **Homogeneous elements**: All elements in the array must be of the same type (e.g., all integers, all floats, etc.).
- **Indexing**: The elements of an array are accessed using an index, where the index typically starts from 0.

Two-Dimensional Arrays

A **two-dimensional array** is an array of arrays, essentially representing a grid or table structure. It's used to store data in rows and columns. In simple terms, it can be thought of as a matrix.

There could be situation where a table of values will have to be stored. Consider the following data table.

	Washing Machine	Air Conditioner	Refrigerator
Salesperson 1	15	5	20
Salesperson 2	10	7	15
Salesperson 3	12	2	13
Salesperson 4	7	6	5

The table contains a total of 12 values. This table can be considered as matrix consisting of four rows and three columns. Each row represents the values of sales by a particular salesperson and each column represents the values of sales of washing machine, air conditioner and refrigerator.

C allows us to define such tables of items by using two-dimensional arrays. The table discussed above can be defined in C as “a[4][3]”.

Two-dimensional arrays are declared as follows:

```
type  array_name [row_size][column_size];
```

Characteristics of Two-Dimensional Arrays:

- It is a collection of data where each element is accessed by two indices: one for the row and one for the column.
- The array can be visualized as a table (or matrix) with rows and columns.
- The size of a two-dimensional array is defined as “number_of_rows×number_of_columns”.

Initializing Two-Dimensional Arrays

Two-dimensional array can be initialized at the time of declaration as given below.

```
int   table[2][3] = {{1,2,3}, {4,5,6}};
```

One can also initialize a two-dimensional array in the form of a matrix as shown below.

```
int  table[2][3] = {
                        {1, 2, 3},
                        {4, 5, 6}
                  };
```

When the array is completely initialized with all the values, one need not specify the size of the first dimension as given below.

```
int  table [ ][3] = {
                        {1, 2, 3},
                        {4, 5, 6}
                  };
```

If the values are missing in an initialize, they are automatically set to zero as given below.

```
int  table [2][3] = {
                        {1, 2},
                        {4}
                  };
```

In above example third element of first row will be initialized as zero and second and third element of second row will be initialized as zero.

4) Write a program that stores roll number and total marks obtained by 65 students in the subject of PPS and display it on the screen.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a[65][2], i;
        clrscr( );
        for (i=1; i<=65; i=i+1)
        {
                printf(“Enter the Roll no. and mark\n”);
                scanf(“%d%d”, &a[i][1], &a[i][2]);
        }
        printf(“Roll no. and mark of students are\n”);
        for (i=1; i<=65; i=i+1)
        {
                printf(“%d%d\n”, a[i][1], a[i][2]);
        }
        getch( );
}
```

Unit-5: Introduction to Python

History and Philosophy of Python

History of Python:

- **Creator**: Python was developed by **Guido van Rossum,** a Dutch programmer in the late 1980s.
- **First Release**: The first version (Python 0.9.0) was released in **February 1991**.
- **Development Philosophy**: Python was designed to emphasize readability and simplicity, which allows developers to write clear, logical code.
- **Name**: Python was named after the British comedy group **Monty Python**, as Guido van Rossum was a fan of their work.
- Python is a successor of the ABC programming language.
- **Python 2 vs Python 3**:
  - **Python 2** was released in **2000** and became the dominant version for years.
  - **Python 3** was introduced in **2008** to fix some inconsistencies in Python 2.
  - **Python 2 End of Life**: Python 2 reached its official end of life on **January 1, 2020**. Python 3 is now the main version.

Philosophy of Python:

The philosophy of Python can be summarized by the **Zen of Python**, which is a set of guiding principles for writing computer programs in Python. It emphasizes:

- **Readability**: Code should be easy to read and understand.
- **Simplicity**: Python's syntax is designed to be simple and straightforward, reducing the complexity of the code.
- **Explicit is better than implicit**: This principle encourages developers to write code that is clear and avoids ambiguity.
- **There should be one—and preferably only one—obvious way to do it**: Python avoids having multiple ways to accomplish the same task, which keeps code consistent.
- **Errors should never pass silently**: Python encourages clear error messages and debugging.

Installing Python

To start programming with Python, you first need to install it on your system. Here's how you can install Python on different platforms:

Installing Python on Windows:

- **Download the installer**: Go to the official Python website "python.org" and download the latest version of Python for Windows.
- **Run the installer**: During installation, make sure to check the box that says "Add Python to PATH" before clicking "Install Now". This ensures that Python is available from the command line.
- **Verify Installation**: Open **Command Prompt**.

  This should display the version of Python you installed, confirming that Python is set up correctly.

First Steps in Python

Once Python is installed, you can start writing and running Python programs. Here are a few basic steps to get started:

1. **Using the Python Interactive Shell**:

- To open the interactive shell (REPL), type python or python3 in your terminal/command prompt:
- This opens an interactive environment where you can immediately execute Python commands.

  Example:
  ```
  >>>print("Hello, World!")
  Hello, World!
  ```

2. **Creating Python Scripts**:

- A Python script is simply a text file with a ".py" extension.
- One can write script (Python files) using IDLE's editor window. Also one can create a script using any text editor (such as Notepad, Visual Studio Code, Sublime Text, etc.). Save the script as file_name.py (".py" is Python file extension)

  Example: hello.py

  For example the script contain following code

```
print("Hello, World!")
```

Note: When you write code in a Python file, you don’t need to include the >>> prompt.

- To open script in IDLE, select “File” and then select “open”.
- To run the program, select “Run” and then select “Run Module” from the menu in the editor window.

3. **Using an Integrated Development Environment (IDE)**:

- For a more robust development environment, you can use IDEs like **PyCharm**, **VSCode**, or **Thonny**, which provide features like code suggestions, debugging tools, and more.

Basic Syntax in Python

Syntax refers to the set of rules that dictate how code must be written to be considered valid and understandable by a computer. It's essentially the grammar and structure of a programming language.

Compilers and interpreters rely on strict adherence to syntax rules to translate human-readable code into machine-executable instructions. Deviations from syntax rules, results in syntax errors and prevent the program from running.

Key aspects of programming syntax include:

1) Statements and indentation:

Python uses indentation (whitespace) to define the structure of the code, such as loops, conditionals, and functions. Unlike other languages that use braces {}, Python relies on indentation to define blocks.

Example:

```
# Correct syntax for an if statement
if x > 10:
    print("X is greater than 10")

# Incorrect syntax (missing colon)
if x > 10
print("X is greater than 10")
```

In this example, the print() statement is indented to indicate it belongs to the if block.

2) Comments:

You can add comments to your code with the # symbol. Comments are ignored during execution.

Example:

```
# This is a single-line comment
print("Hello, World!")  # This is an inline comment
```

3) Variable and Function naming conventions:

Rules for how variables, functions, and other identifiers can be named.

Data Types in Python

Python supports numeric types such as integer, floating point, as well as complex numbers. It also supports text types such as string and sequence type as list, tuple. Further, the value of a variable can be converted from one type to another.

1) Numeric type

- **Integer**: Whole numbers, positive or negative, without decimal

  Example:
  num1 = 10
  num2 = -5

- **Float**: Numbers with decimal points.

  Example:
  pi = 3.14

- **Complex**: Numbers with a real and imaginary part.

  Example:
  complex_num = 2 + 3j

2) Text type:

String: A sequence of characters enclosed in single or double quotes.

Example:
message = "Hello, Python!"
message = 'Hello, Python!'

You can also use triple quotes for multi-line strings:

Example:
multiline_str = """This is
a multi-line string."""

3) Sequence type:

- **List**: Collection of items, which can be changed.

  Example:
  fruits = ["apple", "banana", "cherry"] # list

  Lists can store different data types:

Example:
```
mixed_list = [1, 3.14, "hello", True]
```

- **Tuple**: Collection of items, which can’t be changed.

  Example
  ```
  coordinates = (10, 20)
  ```

Control Structures

Control structures allow us to control the flow of execution in a program. They include conditional statements and loops. Python provides **if**, **if-else**, and **if-elif-else** for decision making.

If Statement

The if statement is used to test a condition, and if the condition is **True**, the block of code inside the if statement is executed.

Example:

```
x = 10
if x >5:
   print("x is greater than 5")
```

If-Else Statement

The else statement follows an if statement and runs if the condition is **False**.

Example:

```
x = 3
if x >5:
    print("x is greater than 5")
else:
   print("x is not greater than 5")
```

If-Elif-Else Ladder

You can use multiple elif (short for "else if") statements to check multiple conditions.

Example:

```
x = 7
if x >10:
   print("x is greater than 10")
elif x == 7:
   print("x is equal to 7")
else:
   print("x is less than 7")
```

In this example, Python will print **"x is equal to 7"** because the second condition is True.

Loops

Loops allow you to repeatedly execute a block of code until a condition is true. Python has **for** and **while** loops.

While Loop

A `while` loop repeatedly executes a block of code as long as the given condition is **True**.

```
Example:
x = 0
while x <5:
    print(x)
    x += 1
```

This will print:

```
0
1
2
3
4
```

For Loop

You can use `for` loops to execute a block of code for each element in the sequence.

```
Example:
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

This will print each element of the `fruits` list:

```
apple
banana
cherry
```

Dictionaries

A **dictionary** in Python is an unordered collection of key-value pairs. It is used to store data that allows quick lookup by key.

Creating a Dictionary

You can create a dictionary using curly braces {} and separating keys from values with a **colon :**.

Example:

```
person = {"name": "Alice","age": 25,"city": "New York"}
```

Accessing Dictionary Values

To access a value in a dictionary, you use its corresponding key.

Example

```
print(person["name"])
Output: Alice
```

Adding or Updating Values

You can add a new key-value pair or update an existing key's value.

Example:

```
person["email"] = "alice@example.com"        # Adding a new key-value pair
person["age"] = 26                           # Updating the value of the existing key "age"
```

Removing Items

You can remove items using the del keyword or the pop() method.

Example:

```
del person["city"]          # Removes the "city" key-value pair
print(person)
```

Functions

A “function” is a block of reusable block of code that is used to perform a specific task. Functions allow you to call the block of code multiple times without rewriting it.

Type of Functions:

Python has two main types of functions as

1) Built-in functions: Pre-defined functions that come with Python. Example: print ( ), len ( ) etc.

2) User-defined functions: Functions that programmers create to perform specific tasks as needed for their applications.

Defining and Calling a Function:

User-defined functions in Python are defined using the “def” keyword. After defining a function, you can call it by using its name followed by parentheses.

Example:

```
# Defining a Function
def  greet(name):
        print(f"Hello, {name}")
        print(“How are you?”)

# Calling s Function
greet(“Mahesh”)
```

Function with Return Value

A function can return a value using the return keyword.

Example:

```
# Defining a Function
def add(a, b):
   z = a + b
   return z

# Calling s Function
result = add(3, 4)
print(result)          # Output: 7
```

Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into "objects," which contain both data (attributes) and methods (functions). Python supports OOP, and it's widely used to create modular, reusable, and organized code.

Need object-oriented programming

- To make the development and maintenance of projects more effortless.
- To provide the feature of data hiding that is good for security concerns.
- We can solve real-world problems if we are using object-oriented programming.
- It ensures code reusability.
- It lets us write generic code: which will work with a range of data, so we don't have to write basic stuff over and over again.

Key Concepts of OOP:

1. **Class**: A blueprint for creating objects. A class defines the data and function common to all objects of that type.

   For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, Car is the class, and wheels, speed limits, mileage are their properties.

2. **Object**: It is a basic unit of Object-Oriented Programming and represents the real-life entities. An instance of a class. It has the data and functions defined in the class.

   For example “Dog” is a real-life Object, which has some characteristics like color, Breed, Bark, Sleep, and Eats.

3. **Encapsulation**: Bundling the data and methods that operate on the data within one unit (class). It restricts direct access to some of the object's components.
4. **Inheritance**: The mechanism by which one class can inherit attributes and methods from another class.
5. **Polymorphism**: The ability of different classes to provide a method with the same name, but with different implementations.

   For example, A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.

6. **Abstraction**: Hiding the complexity and showing only the essential features to the user.

Modules and Libraries

Python modules and libraries allow you to organize and reuse code across different projects. A **module** is a file that contains Python code, and a **library** is a collection of modules that are designed to work together.

Module:

A **module** is simply a Python file that contains functions, classes, and variables. You can use the import statement to include the contents of a module into your current Python script.

```
# Let's say you have a file named `math_operations.py`
# math_operations.py
def add(a, b):
return a + b

# In your main program, you can import the module
import math_operations
print(math_operations.add(5, 3))                    # Output: 8
```

Python comes with many **built-in modules** that provide various functionalities like math operations and more. Some commonly used built-in modules include:

- **math**: Provides mathematical functions like `sin`, `cos`, `sqrt`, and `log`.

  ```
  import math
  print(math.sqrt(16))  # Output: 4.0
  ```

Library

Python has a rich ecosystem of third-party libraries that extend its functionality. You can install these libraries using **pip**, Python's package manager.

1. **Installing Libraries with pip**: You can install an external library using pip from the command line:
2. **Using the Requests Library**: Once installed, you can use the external library in your program:
3. **Popular External Libraries**:
   - **NumPy**: Used for numerical computations and working with arrays.
   - **Pandas**: Data analysis and manipulation.
   - **Matplotlib**: Plotting and visualization.
   - **Flask**: A web framework for building web applications.
   - **TensorFlow / PyTorch**: Libraries for machine learning and deep learning.

Features of Python

- Python is a general-purpose programming language and supports multiple paradigms or ways of programming.
- It can also be used to write functional programs (applying and composing functions), among others.
- . This means Python programs are not compiled and stored into a binary code file (e.g., an executable), but its source is translated into machine code and executed by the interpreter directly (without us seeing the executable code).
- Python is also dynamically-typed. This means that the type of a variable may not be specified in the source code, and is identified when the program executes.
- Python also has a large variety of standard libraries, which allow us to write complex codes quickly, improving our productivity.

IDLE: Integrated Development and Learning Environment
IDE: Integrated Development Environment
REPL: Read, Evaluate, Print Loop