1.1 Programming Languages

A programming language is a formal system designed to communicate instructions to a computer. It is a set of rules and syntax that allows programmers to write code to instruct a computer to perform specific tasks. Programming languages are used to develop software, applications, and other types of computer programs.

Types of programming language

1. Low-level programming language

Low-level language is **machine-dependent (0s and 1s)** programming language. The processor runs low-level programs directly without the need of a compiler or interpreter, so the programs written in low-level language can be run very fast.

Low-level language is further divided into two parts -

i. Machine Language

Machine language is a type of low-level programming language. It is also called as **machine code or object code**. Machine language is easier to read because it is normally displayed in binary or hexadecimal form (base 16) form. It does not require a translator to convert the programs because computers directly understand the machine language programs.

The advantage of machine language is that it helps the programmer to execute the programs faster than the high-level programming language.

ii. Assembly Language

Assembly language (ASM) is also a type of low-level programming language that is designed for specific processors. It represents the set of instructions in a **symbolic and human-understandable form**. It uses an assembler to convert the assembly language to machine language.

The advantage of assembly language is that it requires less memory and less execution time to execute a program.

2. High-level programming language

High-level programming language (HLL) is designed for **developing user-friendly software programs and websites**. This programming language requires a compiler or interpreter to translate the program into machine language (execute the program).

The main advantage of a high-level language is that it is **easy to read, write, and maintain**.

High-level programming language includes **Python, Java, JavaScript, PHP, C#, C++, Objective C, Cobol, Perl, Pascal, LISP, FORTRAN, and Swift programming language**.

A high-level language is further divided into three parts -

i. Procedural Oriented programming language

Procedural Oriented Programming (POP) language is derived from structured programming and based upon the procedure call concept. It divides a program into small procedures called **routines or functions**.

Procedural Oriented programming language is used by a software programmer to create a program that can be accomplished by using a programming editor like IDE, Adobe Dreamweaver, or Microsoft Visual Studio.

The advantage of POP language is that it helps programmers to easily track the program flow and code can be reused in different parts of the program.

The advantage of POP language is that it helps programmers to easily track the program flow and code can be reused in different parts of the program.

Example: C, FORTRAN, Basic, Pascal, etc.

ii. Object-Oriented Programming language

Object-Oriented Programming (OOP) language is **based upon the objects**. In this **programming language, programs are divided into small parts called objects**. It is used to implement real-world entities like inheritance, polymorphism, abstraction, etc in the program to makes the program resusable, efficient, and easy-to-use.

The main advantage of object-oriented programming is that OOP is faster and easier to execute, maintain, modify, as well as debug.

Note: Object-Oriented Programming language follows a bottom-up approach.

Example: C++, Java, Python, C#, etc.

3. Middle-level programming language

Middle-level programming language **lies between the low-level programming language and high-level programming language**. It is also known as the intermediate programming language and pseudo-language.

A middle-level programming language's advantages are that it supports the features of high-level programming, it is a user-friendly language, and closely related to machine language and human language.

Example: C, C++, language

1.2 Software and its types

In the current world, we as a user are blown away with modern software. They have surrounded us to make our lives better. Therefore, knowingly or unknowingly, we use different software to lead our day-to-day activities efficiently and more reliably. Furthermore, with time, people are becoming more tech-savvy, and that's why the need for high-tech technologies and software support is constantly changing and increasing with the growing needs and demands of people. Today in the market, there are various types of software, which can be valuable for any user, even those who don't recognize these diverse types of software. Software has such a powerful impact on our lives, but many of you might be thinking, "what does software means" or "what are its various types that exist today?"

Software is a set of programs (sequence of instructions) that allows the users to perform a well-defined function or some specified task."

Software is responsible for directing all computer-related devices and instructing them regarding what and how the task is to be performed. However, the software is made up of binary language (composed of ones and zeros), and for a programmer writing the binary code would be a slow and tedious task. Therefore, software programmers write the software program in various human-readable languages such as Java, Python, C#, etc.

Types of Software

Software's are broadly classified into two types, i.e., **System Software and Application Software**.

1. System Software

System software **is a computer program that helps the user to run computer hardware or software and manages the interaction between them**. Essentially, it is software that **constantly runs in the computer background, maintaining the computer hardware and computer's basic functionalities, including the operating system, utility software, and interface**. In simple terms, you can say that the system acts as a middle man that checks and facilitates the operations flowing between the user and the computer hardware.

System software is not limited to the operating system. They also include the basic I/O system procedures, the boot program, assembler, computer device driver, etc. This software supports a high-speed platform to provide effective software for the other applications to work in effortlessly. Therefore system software is an essential part of your computer system. **They are the first thing that gets loaded in the system's memory wherever you turn on your computer**. **System software is also known as "low-level software**" because the end-users do not operate them. Companies usually employ the best software development programmers who can deploy efficient system software.

The further classifications of system software are as follows:

1. Operating System

The operating system is the most prominent example of system software that acts as an interface between the user and system hardware. **It is a group of software that handles the execution of programs and offers general services for the application that runs over the computer.** There are various types of operating systems available in the market, such as embedded operating systems, real-time OS, distributed OS, single or multi-user operating system, mobile, Internet, and various others.

Some of the commonly used examples of operating systems are given below.

- Microsoft Windows
- Apple's iOS
- Apple's MacOS
- Android
- CentOS
- Linus
- Ubuntu
- Unix

2. Device Drivers

In computing, **the device driver is a type of software that operates or controls some specific hardware devices linked to your system**. They provide a software interface to hardware devices allowing computer operating systems and other applications to fetch hardware functions without knowing the exact specifications of the hardware. Some common examples of such device drivers that connect hardware devices (printers, sound cards, network cards, hard disks, floppy disk, keyboard, mouse, etc.) to a system easily are as follows:

- BIOS (Basic Input/Output System) Device Driver
- USB (Universal Serial Bus) Drivers
- Motherboard Drivers
- Display Drivers
- Printer Drivers
- Sound Card Driver
- ROM (Read-only memory) Drivers
- VGA (Video Graphic Array) Drivers

3. Firmware

Firmware **is a type of permanent software embedded in the system's ROM (read-only memory) to provide low-level control for some particular system device hardware**. It is a set of instructions that are stored permanently on your computer's hardware device.

Common examples of devices utilizing firmware are given below:

- Computer Peripherals
- Consumer Appliances
- Embedded Systems
- UEFI (United Extensible Firmware Interface)
- BIOS (Basic Input/Output System)

4. Utility

Utility software is developed to provide support in analyzing, optimizing, along configuring and maintaining a computer. The job of the utility program is to offer support to the system infrastructure. Though the system will work even if it doesn't have any utility software, the right kind of utility software enhances its performance and makes it more reliable.

Some of the common examples of utility software are as follows:

- Norton and McAfee Antivirus
- WinRAR
- Directory Opus
- Disk defragmenter
- WinZip
- Windows File Explorer
- Razer Cortex

Application Software

Application programs are end-user computer programs developed primarily to provide specific functionality to the user. The applications programs assist the user in accomplishing numerous tasks such as doing **online research, completing notes, designing graphics, managing the finances, watching a movie, writing documents, playing games, and many more**. Therefore, many software applications are designed and developed every year by companies as per the demand and requirements of the potential users. The application software can either be designed for a general-purpose or specially coded as per the requirements of business cooperation.

Today there are varieties of application software available in the market. Given below are some of the popular examples:

a. Word Processors

Word processor applications are globally **used for documentation, making notes, and typing data.** It also helps the end-users store and format data. They also enable the users to print their documents.

Some examples of Word Processor software's are as follows:

- MS Word (Microsoft)
- iWork-Pages (Apple)
- Corel WordPerfect
- Google Docs

b. Database Software

Database software is **used to create, manage, modify and organize a massive amount of data quickly retrieved.** Another name for database software **is Database Management System (DBMS).** Such software helps companies in their data organization. Common examples of Database Software's are:

- Oracle
- MS Access
- SQLite
- Microsoft SQL Server
- FileMaker
- dBase
- MariaDB
- MySQL

c. Multimedia Software

This software **enables the users to play, create or record images, music, and video files.** Different graphic designing companies widely use multimedia software to make animation, images, posts, packaging, marketing creative, gif, or even video editing. Due to their popularity and increasing demand, every software product development corporation has massive avenues in creating and upgrading them.

Common examples of Multimedia Software's are given below:

- Adobe Photoshop
- Windows Movie Maker
- Adobe Illustrator
- Picasa
- Windows Media Player
- Corel Draw

d. Web Browsers

A web browser is an application for accessing websites and the Internet.[1] When a user requests a web page from a particular website, the browser retrieves its files from a web server and then displays the page on the user's screen. Browsers are used on a range of devices, including desktops, laptops, tablets, and smartphones. These are a type of software that is globally used to browse the Internet. **Web browsers help the users in positioning as well as fetching data across the web.** Common examples of web browsers are given below:

- Chrome
- Mozilla Firefox
- Microsoft Internet Explorer
- Opera
- Microsoft Edge
- UC Browser
- Apple Safari

1.3 Structured Programming

Structured programming, or modular programming, is a programming paradigm that facilitates the creation of programs with readable code and reusable components. All modern programming languages support structured programming.

The key principles of structured programming include:

1. **Sequential Control Flow:**
    - Programs are organized as sequences of statements executed one after the other.
    - Execution flows sequentially from the beginning of the program to the end.
2. **Selection or Decision Structures:**
    - Conditional statements, such as "if-else" or "switch," are used to make decisions based on certain conditions.
    - These structures allow the program to take different paths depending on whether a condition is true or false.
3. **Repetition or Iteration Structures:**
    - Looping constructs, like "for," "while," or "do-while," are employed for repetitive tasks.
    - These structures enable the execution of a set of statements multiple times until a specified condition is met.
4. **Modular Design:**
    - Programs are divided into smaller, manageable modules or functions.
    - Each module serves a specific purpose and can be independently developed, tested, and maintained.

The benefits of structured programming include:

- **Readability and Maintainability:** The use of clear and well-defined structures makes code easier to read and maintain. Each module can be understood in isolation, making it easier for programmers to collaborate and update code.
- **Debugging and Testing:** The modular nature of structured programming simplifies the process of identifying and fixing errors. Individual modules can be tested independently, making it easier to locate and correct issues.
- **Controlled Complexity:** Breaking down a program into smaller, more manageable modules helps control the complexity of the code. Each module focuses on a specific task, making it easier to understand and reason about the program's behavior.
- **Reusability:** Modular design allows for the reuse of modules in different parts of a program or in other projects, promoting a more efficient development process.

Structured programming is in contrast to unstructured programming, where code may be written as a series of jumps and branches without clear modularization. The principles of structured programming have been widely adopted and influenced the design of many programming languages, contributing to the development of more maintainable and scalable software systems.

1.4 Problem Solving using Computer

Sometimes it is not sufficient just to cope with problems. We have to solve that problems. Most people are involving to solve the problem. These problem are occur while performing small task or making small decision.

Problem-solving using a computer is a fundamental skill in computer science and programming. It involves a combination of logical thinking, algorithmic design, coding proficiency, and the ability to analyze and understand the problem domain. This process is applicable in various fields, from software development and data analysis to scientific research and automation.

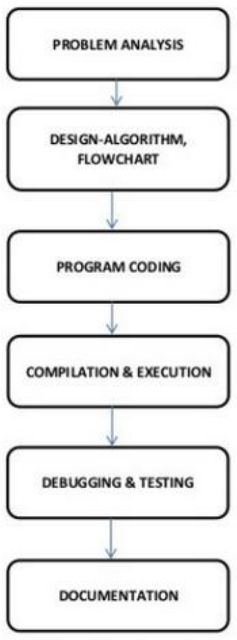


Figure: Problem solving process

Here are the key components of problem-solving using a computer:

1. **Problem Analysis:**
   - Begin by clearly understanding the problem you need to solve.
   - Identify the input requirements, expected outputs, and any constraints.
2. **Design Algorithm and Flowchart:**
   - Develop a step-by-step algorithm to solve the problem.
   - Create a flowchart that visually represents the logical flow of the algorithm, including decision points and loops.
3. **Program Coding:**
   - Translate the algorithm into a programming language (e.g., Python, C++, Java).
   - Write the code, ensuring it follows the syntax and conventions of the chosen language.
4. **Compilation and Execution:**
   - Use a compiler or interpreter to translate the human-readable code into machine-readable instructions.
   - Execute the program to observe its behavior.
5. **Debugging and Testing:**
   - Identify and fix any errors or bugs in the code.
   - Test the program with various inputs to ensure it produces the correct outputs.
   - Use debugging tools to step through the code and pinpoint issues.
6. **Documentation:**
   - Provide comprehensive documentation that includes:
     - Description of the problem the program solves.
     - Details of the algorithm used.
     - Instructions for using the program.
     - Comments within the code to explain complex sections.

Problems Analysis

Problem analysis is a critical phase in the software development process that involves a systematic examination of a problem to be solved by a computer program. This analysis includes:

1. **Understanding of the Problem:**
   - Involves comprehending the nature and context of the problem.
   - Identifying the key stakeholders, their needs, and the goals of the solution.
   - Defining the problem's scope and boundaries.
2. **Feasibility Analysis:**
   - Assessing the feasibility of solving the problem with a computer program.
   - Evaluating technical, economic, operational, and scheduling feasibility.
   - Determining if the proposed solution aligns with the organization's capabilities and resources.
3. **Requirement Analysis:**
   - Identifying and documenting detailed requirements for the solution.
   - Engaging with stakeholders to gather and prioritize their needs.
   - Defining functional and non-functional requirements that the program must fulfill.

Design

Designing a program involves creating a plan or blueprint that outlines the structure and logic of the software before the actual coding takes place. This process includes defining algorithms and using flowcharts to visually represent the steps and decision-making within those algorithms.

Algorithms

Algorithm is the set of rules that define how particular problem can be solved in finite number of steps. Any good algorithm must have following characteristics:

1. **Input:** Specify and require input
2. **Output:** Solution of any problem
3. **Definite:** Solution must be clearly defined
4. **Effective**
5. **Finite:** Steps must be finite
6. **Correct:** Correct output must be generated

Advantages of Algorithms:

1. It is the way to sole a problem step-wise so it is easy to understand.
2. It uses definite procedure.
3. It is not dependent with any programming language.
4. Each step has it own meaning so it is easy to debug.

Disadvantage of Algorithms:

1. It is time consuming.
2. Difficult to show branching and looping statement.
3. Large problems are difficult to implement.

Flowchart:

The solution of any problem in picture form is called flowchart. It is the one of the most important technique to depict an algorithm.

Advantage of Flowchart:

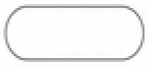
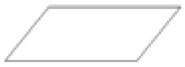
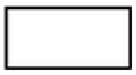
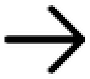
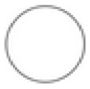
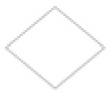
1. Easier to understand
2. Helps to understand logic of problem
3. Easy to draw flowchart in any software like MS-Word
4. Complex problem can be represent using less symbols
5. It is the way to documenting any problem
6. Helps in debugging process

Disadvantage of Flowchart:

1. For any change, Flowchart have to redrawn
2. Showing many looping and branching become complex
3. Modification of flowchart is time consuming

Symbol Used in Flowchart:

Symbol	Name	Description

	Terminal	Terminal represent start and end
	Input / Output	Used for input (reading) and output (printing) operation.
	Processing	Used for data manipulation and data operations.
	Arrow	Used to represent flow of operations.
	Connector	Used to connect different flow of lines
	Decision	Used to make decision

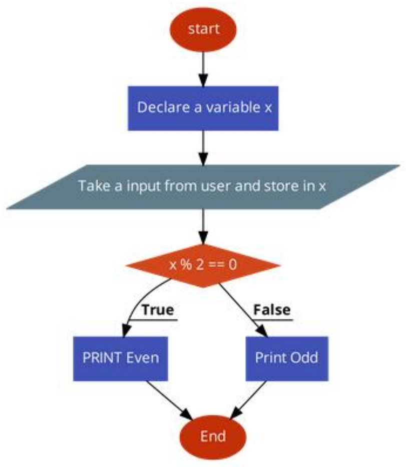
Example: Algorithm and Flowchart to check odd or even

Solution:

Algorithm:

```
Step 1: Start
Step 2: Declare a variable x
Step 3: Take a input from user and  store in x
Step 4: IF x % 2 == 0 THEN
            PRINT Even
        ELSE
            PRINT Odd
Step 5: End
```

Flowchart:



Coding

Coding, also known as programming, is the process of writing instructions for a computer to perform specific tasks. These instructions are written in a programming language, and the resulting set of instructions is called code. A programmer or developer creates code to solve a problem, build software, or implement a specific functionality. The coding process involves translating a solution or algorithm into a format that a computer can understand and execute.

1. **Compilation/Interpretation:**
   - **Compilation:**
     - Compilation is the process of translating the entire source code of a program into machine code or an intermediate code.
     - The result of compilation is often an executable file that can be run independently of the original source code.
     - Compiled languages, such as C or C++, produce machine code specific to the target platform during compilation.
   - **Interpretation:**
     - Interpretation involves translating the source code into machine code or executing it directly without creating a separate executable.
     - Interpreted languages, such as Python or JavaScript, are executed line by line by an interpreter, and the source code remains in its original form.
     - Some languages, like Java, use a combination of compilation and interpretation, where code is initially compiled into an intermediate form (bytecode) and then interpreted by a virtual machine.
2. **Execution:** It refers to the process of running a computer program to carry out the instructions and tasks defined in its source code. Once a program has been written and translated into machine code (either through compilation or interpretation), it is ready for execution.
   - **Compiled Languages:**
     - In the case of compiled languages, the compiled executable file is executed directly by the computer's hardware or operating system.
     - The compiled code typically runs faster than interpreted code as it has already been translated into machine-specific instructions.
   - **Interpreted Languages:**
     - In interpreted languages, the source code is executed by an interpreter during runtime.
     - The interpreter reads the code line by line and converts it into machine code.
     - Interpreted languages are often platform-independent, but they might have slower execution compared to compiled languages.

Testing and debugging

Testing is the process of evaluating a software application to identify defects or errors, ensuring that it meets specified requirements. Types of errors include syntax errors (grammatical mistakes in code), runtime errors (occur during program execution), and logical errors (flaws in program design).

Types of Errors:

1. Syntax Errors: Violations of programming language rules, detected during compilation.
2. Runtime Errors: Occur during program execution, often due to unexpected input or conditions.

3. Logical Errors: Flaws in the program's design, leading to unintended behavior.

Debugging: Debugging is the systematic process of finding, isolating, and fixing defects or bugs in software code. It involves identifying, understanding, and resolving issues to ensure the correct functioning of the program.

Testing Purposes: Testing is usually performed for the following purposes:

1. **Identifying Defects:** To discover and rectify errors or bugs in the software.
2. **Verification:** To ensure that the software meets specified requirements and performs as expected.
3. **Validation:** To validate that the software satisfies the needs of the end-users or stakeholders.
4. **Quality Assurance:** To maintain and improve the overall quality and reliability of the software.
5. **Risk Mitigation:** To reduce the risk of software failures or malfunctions in production.
6. **Documentation:** To provide evidence of the software's correctness and adherence to requirements.

Implementation

Implementation refers to the stage where the design of the software is translated into a programming language, and the actual code is written and tested. It is the phase where developers bring the software design to life by creating the executable program that will run on a computer or other computing devices.

Evaluation and Maintenance of computer programs

Evaluation involves assessing a computer program's functionality, performance, usability, security, compatibility, and scalability to ensure it meets specified requirements and user expectations.

Maintenance encompasses corrective, adaptive, perfective, and preventive activities to address defects, adapt to changes, enhance functionality, prevent issues, and ensure ongoing reliability and effectiveness of computer programs. This includes updating documentation, implementing version control, monitoring performance, and planning for backup and recovery.

Program documentation

Program documentation starts from the starting of the software development life cycle (SDLC). It keeps most of the information of all phases while developing projects. Documentation is used for future reference for both original programmer and the beginner. The final documentation should contain the following information:

- The program analysis document with objectives input, output, and processing requirements. This is also called Software Requirement Specification (SRS).
- Program design document, algorithm and detailed flowchart and other appropriate diagrams.
- Program verification documents with details of checking, testing and correction procedure along with the list of test data.
- Log is used to document future program revision and maintenance activity.