BCA (Honours)

Fundamentals of Programming Using C

Module 1

Problem-Solving Life Cycle

The **problem-solving life cycle** is a structured process used by developers to identify, analyze, and solve problems in programming or software development. This life cycle ensures that the problem is understood thoroughly, a solution is planned effectively, and the solution is implemented in an organized manner. In order to solve a problem by the computer, one has to pass though certain stages or steps.

1. Understanding the Problem Statement
2. Analyzing the Problem

Understanding the Problem Statement

The first and most crucial step in the problem-solving life cycle is understanding the problem statement. A clear understanding of the problem ensures that the solution developed addresses the real issue.

Objectives:

- To determine **what the problem is**.
- To identify the **goals** that need to be achieved by solving the problem.
- To define the **scope** of the problem, ensuring that unnecessary complexity is avoided.

Steps to Understand the Problem:

1. **Reading the Problem Statement**:
    - Carefully read the entire problem statement.
    - Highlight any key information, requirements, and constraints.

2. **Identify Inputs and Outputs**:
    - Determine what inputs are provided and what outputs are expected.
    - Example: In a program that calculates the area of a rectangle, the **inputs** would be the length and width, and the **output** would be the area.
3. **Understand Constraints**:
    - Identify any restrictions or limitations in the problem. For example, are there limits on the size of inputs? Is there a maximum execution time?
4. **Clarify Ambiguities**:
    - If the problem statement is unclear or ambiguous, clarify it with stakeholders or users to avoid any misinterpretation.

Example:

Suppose you are tasked with writing a program that calculates the total price of items in a shopping cart, applying discounts when applicable.

- **Problem Statement**: Write a program that reads the price of each item in a shopping cart and applies a 10% discount to items that cost more than Rs. 1000. The program should output the total price after all discounts are applied.
- **Inputs**: Prices of items.
- **Outputs**: Total price after applying discounts.
- **Constraints**: Only items costing more than Rs. 1000 will receive a discount.

Understanding this clearly is the first step toward building the right solution.

Analyzing the Problem

Once the problem is understood, the next step is analyzing it to identify the root cause and devise an approach to solve it efficiently. This stage involves breaking the problem down into smaller, more manageable components.

Objectives:

- To break the problem into smaller, simpler subproblems.
- To understand the logical flow of the solution.
- To ensure that all scenarios and edge cases are considered.

Steps to Analyze the Problem:

1. **Break the Problem into Smaller Tasks**:
    - Divide the problem into several subtasks.
    - Example: In the shopping cart problem, one subtask could be determining if a discount applies, while another could be calculating the total price.
2. **Identify Key Processes**:
    - Analyze what processes or steps need to be followed to transform inputs into the required outputs.
    - For the shopping cart problem, processes might include reading prices, checking if a discount applies, calculating the discounted price, and summing all the prices.
3. **Choose Appropriate Data Structures**:
    - Select the best data structures to represent the problem's components. For example, an array or list could store the prices of the items in the cart.
4. **Identify Special Cases and Edge Cases**:
    - Consider any special or edge cases that could affect the solution. For example, in the shopping cart problem, what happens if there are no items in the cart? What if all items are under Rs. 1000?
5. **Define the Algorithm**:
    - Based on the analysis, define the algorithm that will solve the problem step-by-step. This includes designing the logic that the program will follow.
6. **Check Feasibility**:
    - Ensure that the proposed solution is feasible given the constraints. Does it work for all cases? Is it efficient enough for large inputs?

Example:

For the shopping cart problem:

- **Subtasks**:
    1. Read the prices of items from the user.
    2. Check if each price is greater than Rs. 1000 and apply a 10% discount if so.
    3. Sum the total prices after applying discounts.
- **Edge Cases**:
    1. What if there are no items in the cart?
    2. What if all items are below Rs. 1000?

- **Algorithm**:
  1. Start.
  2. Read the prices of all items in the cart.
  3. For each item, if the price is greater than Rs. 1000, apply a 10% discount.
  4. Sum the total prices of all items.
  5. Output the total price.
  6. End.

Planning Program design using Hierarchy charts

Program design consists of the steps a programmer should do before they start coding the program in a specific language. These steps when properly documented will make the completed program easier for other programmers to maintain in the future. There are three broad areas of activity:

- Understanding the Program
- Using Design Tools to Create a Model
- Develop Test Data

Hierarchy Charts

A **Hierarchy Chart**, also known as a **structure chart**, visually represents the organization of a program. It breaks down the program into modules or functions, showing their relationships in a hierarchical manner.

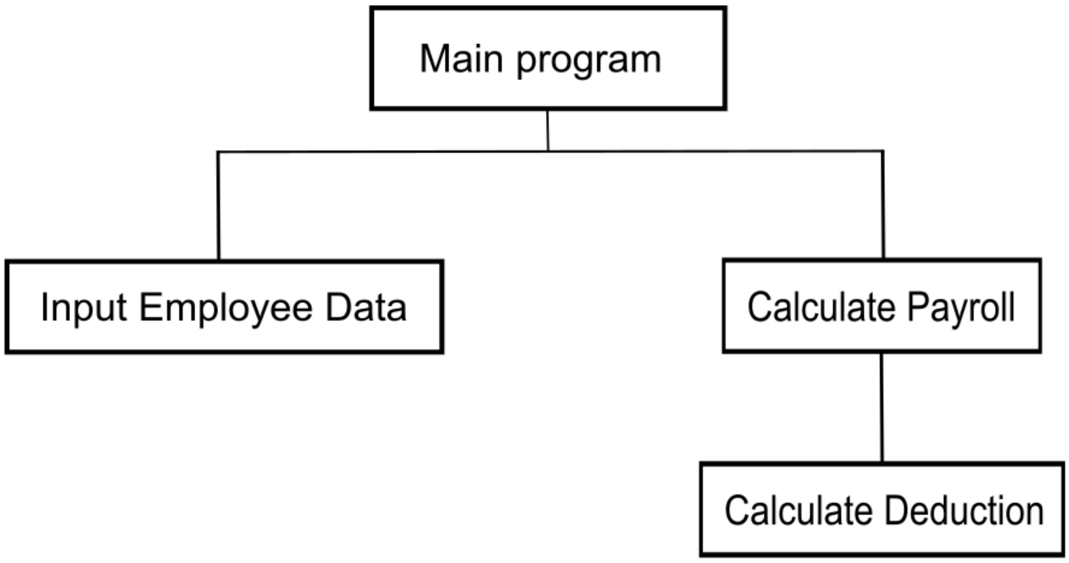
Characteristics:

- A hierarchy chart shows how different modules are related.
- It illustrates the flow of control in a program.
- Helps in understanding how high-level tasks are divided into smaller sub-tasks.

Structure of a Hierarchy Chart:

- At the top level, the main module (or function) is shown.
- Below the main module, submodules are displayed, showing the decomposition of tasks.
- Each module performs a specific task and may call submodules.

A program to calculate the payroll for employees could have a hierarchy chart as follows:



- The **Main Program** handles the overall control.
- It calls two submodules: **Input Employee Data** and **Calculate Payroll**.
- **Calculate Payroll** calls a further submodule, **Calculate Deductions**.

Benefits:

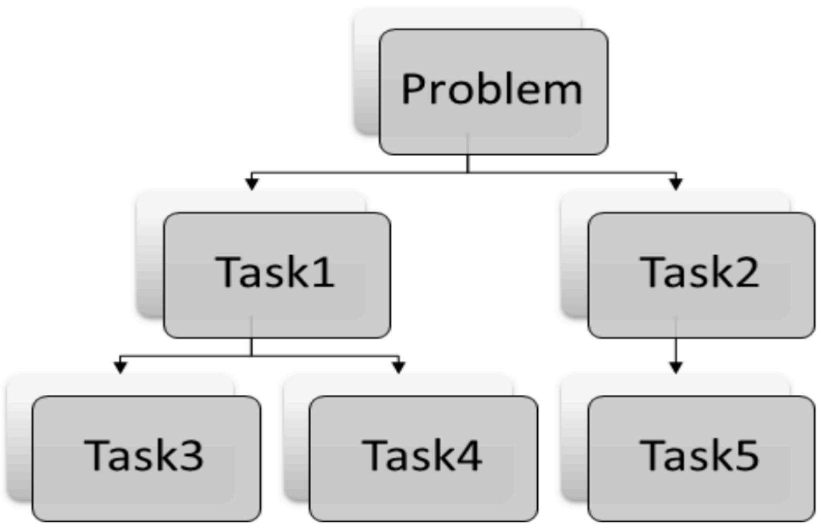
- Provides a clear overview of the program's structure.
- Simplifies complex programs by dividing them into smaller, manageable tasks.
- Makes it easier to assign tasks to teams or individuals during development.

Top-down Approach

The **Top-down approach** is a program design strategy where the overall system is defined first, and then it is broken down into smaller, more detailed components. The focus starts at the high level, gradually moving towards the finer details.

In this approach, the given problem is divided into two or more sub problems, each of which resembles the original problem. The solution of each sub problem is taken out independently.

Finally, the solution of all sub problems is combined to obtain the solution of the main problem. The following figure shows the meaning of top down approach.

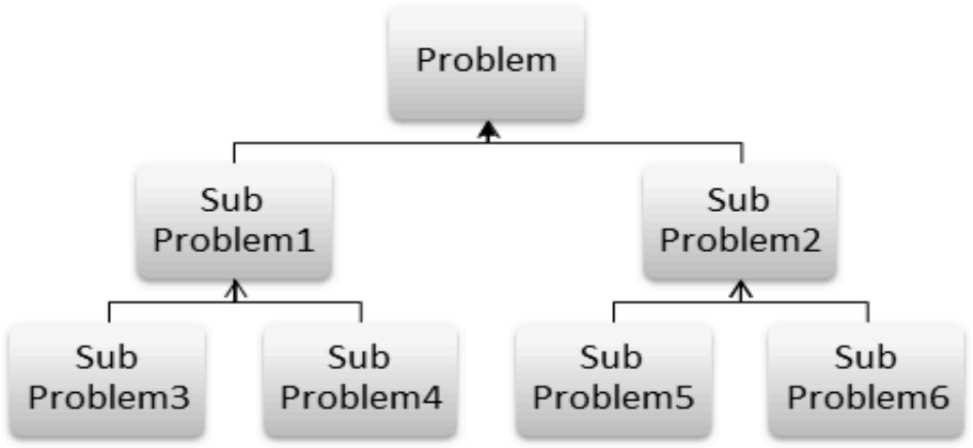


Characteristics of Top-down Approach:

- Starts by identifying the main problem or system and then decomposes it into smaller subproblems.
- Each subproblem is further broken down until the solution becomes simple enough to implement.
- Focuses on modular design, ensuring that each module handles a specific task.

Bottom-up Approach

This technique is just reverse of the top down programming. In this programming technique, the solutions of the independent sub-problems are designed first. Then these solutions are combined or composed in a main module in order to design the final solution of the problem. The following figure shows the bottom-up approach.



Characteristics of Bottom-up Approach:

- Focuses on solving the basic, low-level problems first.
- Once the lower-level modules are developed, they are combined to build higher-level modules.
- Often used in conjunction with object-oriented programming (OOP) principles where reusable components are built first.

Understanding basic Problem-Solving Tools

Algorithms

An algorithm is a step-by-step, well-defined procedure or set of instructions designed to solve a problem or perform a specific task. It is the blueprint for writing the code. Algorithms are used to plan out the logic before the actual coding process begins.

Attributes of an Algorithm:

1. **Input**: Algorithms take zero or more inputs from external sources.
2. **Output**: Every algorithm must provide at least one output, which is the solution to the problem.
3. **Finiteness**: The algorithm must terminate after a finite number of steps.
4. **Definiteness**: Each step of the algorithm must be clear and unambiguous.
5. **Effectiveness**: The steps of the algorithm must be simple enough that they can be carried out, in a finite amount of time, with available resources.
6. **Generality**: The algorithm should be applicable to all problems of a particular class, not just one specific instance.

Example:

An algorithm for finding the sum of two numbers could be:

1. Start
2. Input two numbers (A and B)
3. Add the numbers and store the result in a variable called "Sum"
4. Output the value of "Sum"
5. Stop

Flow Chart

A **flowchart** is a graphical representation of an algorithm or process. It uses symbols and arrows to depict the flow of control in a system or algorithm, showing the sequence of steps to solve a problem. Flowcharts help visualize the structure of an algorithm before implementation.

Attributes of a Flowchart:

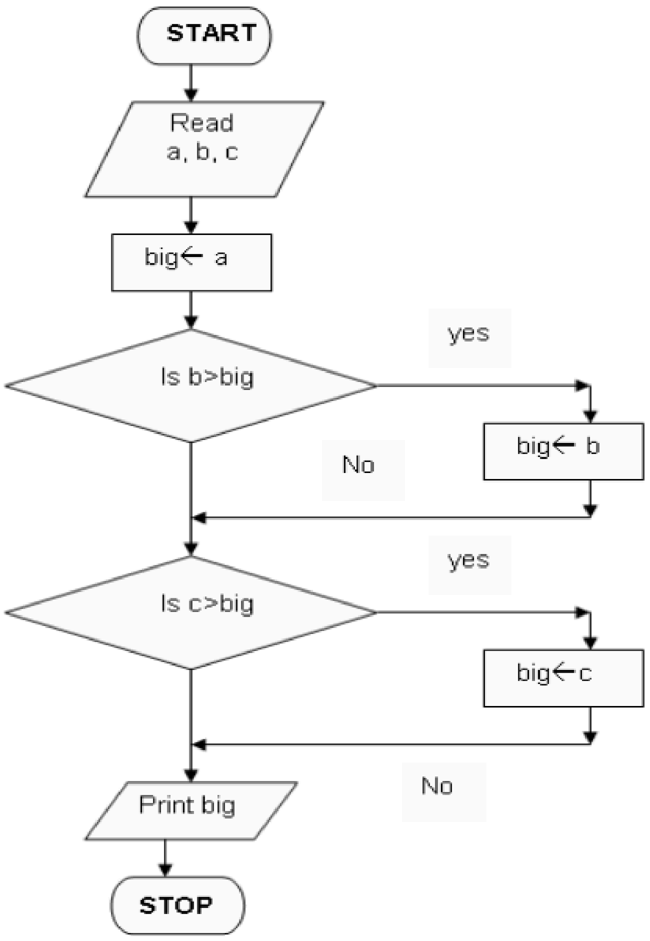
1. **Flow**: It shows the flow of control from one process to another, following a logical sequence.
2. **Clarity**: The symbols and flow should be clear and easily understandable.
3. **Modularity**: Flowcharts often break down complex problems into smaller, manageable parts, which can be represented in separate flowcharts.
4. **Symbols**: Various symbols represent different types of operations or decisions in the process.

Common Flowchart Symbols:

Symbol	**Meaning**
Oval (Start/End)	Indicates the beginning or the end of a process.
Parallelogram	Used to represent input/output operations (e.g., read data, display output).

Rectangle (Process)	Represents a process, such as calculations or instructions.
Diamond (Decision)	Used for decision-making operations (e.g., yes/no, true/false conditions).
Arrow	Shows the direction or flow of control in the process.

The following flowchart is for the program for finding the greatest number from given three numbers.



Statements

Input-Output Statements

These are the basic statements that handle the input and output operations in a program.

- **Input Statements**: Commands that allow the user to provide input to the program.
  - Example in C: `scanf("%d", &num);` for reading an integer.
- **Output Statements**: Commands that display information to the user.
  - Example in C: `printf("Sum is %d", sum);` for printing the result.

Decision-Making Statements

Decision-making statements allow the program to make choices based on certain conditions. The flow of execution can change based on whether a condition evaluates to true or false.

Common Types of Decision-Making Statements:

- **if statement**: Executes a block of code if a condition is true.
- **if-else statement**: Executes one block if the condition is true, and another block if it is false.
- **switch statement**: Allows multiple possible execution paths based on the value of an expression.

Looping Statements

Looping statements allow a set of instructions to be executed repeatedly based on a condition or for a specified number of iterations.

Types of Looping Statements:

- **for loop**: Used when the number of iterations is known beforehand.
- **while loop**: Repeats a block of code while a condition is true.
- **do-while loop**: Similar to a while loop, but the condition is checked after the first iteration.

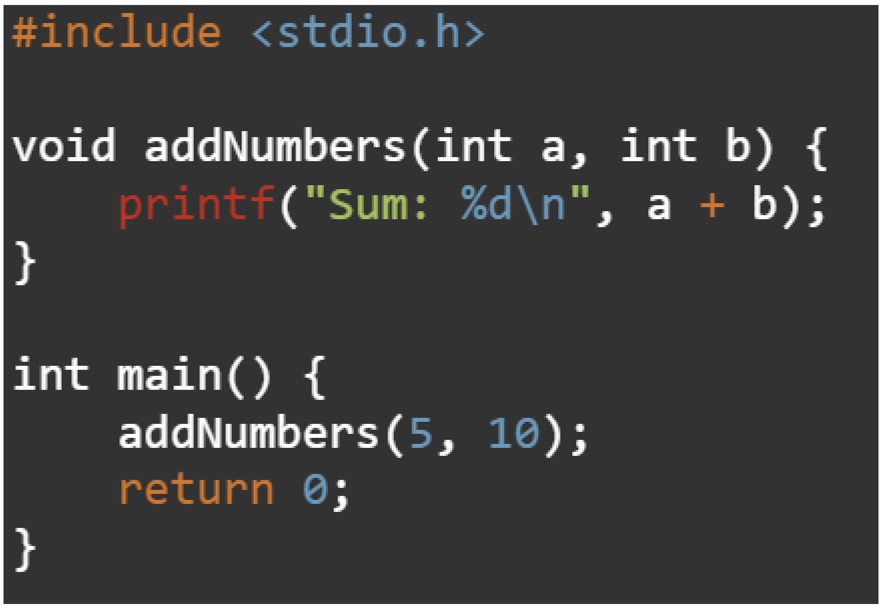
Module Representation

Modules represent the division of a program into separate functions or procedures, each performing a specific task. This approach promotes **modularity**, making the program easier to manage, test, and maintain.

Attributes of Module Representation:

- **Encapsulation**: Each module contains its own data and logic.
- **Reusability**: Modules can be reused in different parts of the program or in different programs.
- **Abstraction**: Modules hide the complexity of implementation from other parts of the program.

Example (in C):



```c
#include <stdio.h>

void addNumbers(int a, int b) {
    printf("Sum: %d\n", a + b);
}

int main() {
    addNumbers(5, 10);
    return 0;
}
```

In this example, `addNumbers` is a module (function) that performs the task of adding two numbers and printing the result. The `main` function calls this module to solve part of the problem.

Introduction to Programming

Computer Program

A **computer program** is a set of instructions written in a programming language that a computer executes to perform a specific task or set of functions. A program acts as a bridge between the computer hardware and the user, instructing the computer how to process input data and generate the desired output.

Classification of Computer Languages

Programming languages allow humans to write instructions in a way that a computer can understand and execute. Languages are classified based on their level of abstraction from the hardware and how they are translated into machine code.

Programming languages can be classified into three categories as follows:

1. Machine Language
2. Assembly Language
3. High-Level Language

Machine Language

Machine language (or machine code) is the most basic programming language, consisting of binary digits (0s and 1s). This is the only language directly understood by the computer's CPU.

Characteristics:

- **No Translation Required**: Machine language does not need to be translated, as it is directly executed by the CPU.
- **Hardware-Specific**: Machine code is specific to the CPU architecture (e.g., x86, ARM).
- **Difficult to Understand**: Writing programs in machine language is complex, as the programmer must manage memory addresses, CPU registers, and instruction sets.

Assembly Language

A low-level language that uses symbolic representations (mnemonics) for machine instructions, making it slightly easier for humans to work with. These low-level languages are very close to the way machines communicate but do not reach binary. The disadvantage of these languages is that they are specific to each machine. Example: Assembly language.

Characteristics:

- Requires an **assembler** to convert the assembly code into machine code.
- Still hardware-specific, but more readable than machine language.
- Easier to debug than machine language but still challenging compared to high-level languages.

High-Level Language

High-level languages are closer to human languages and provide greater abstraction from the hardware. They allow programmers to write complex algorithms without worrying about the underlying machine architecture. Example: C, C++, Java, etc

Characteristics:

- **Translator Required**: High-level languages require a **compiler** or **interpreter** to translate the code into machine language.
- **Portable**: High-level languages are hardware-independent, meaning the same code can run on different types of hardware with minimal changes.
- **Easy to Read and Write**: These languages use human-readable syntax, making programming more accessible and reducing development time.

Language Translator

Language translators are essential tools that convert high-level or assembly language programs into machine language so that computers can understand and execute them. Without translators, programs written by humans in various languages would be useless to the computer hardware.

There are three primary types of language translators:

1. Assembler
2. Compiler
3. Interpreter

Assembler

An **assembler** is a translator that converts **assembly language** programs into machine language. Assembly language is a low-level language that uses mnemonics (instructions) to represent machine instructions. Since assembly language is still quite close to machine language but more readable for humans, the assembler's job is relatively straightforward compared to other translators.

How an Assembler Works:

- The assembler reads the assembly code line by line and converts each mnemonic into its equivalent machine instruction.
- It also handles any **symbolic references** (labels, variables) in the program by translating them into actual memory addresses.
- The output of an assembler is an **object file**, which contains the translated machine code, ready for execution by the CPU or further processing by a **linker**.

Example of Assembly Code:

```
MOV AX, 5;    -- Move the value 5 into register AX

ADD AX, BX;  -- Add the value in register BX to AX
```

Compiler

A **compiler** is a program that translates a high-level language (HLL) into machine language. The translation process happens all at once, producing a complete **machine code** program (often referred to as an **object code** or **executable**).

In a compiler, the source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of the compilation with line numbers when there are any errors in the source code. The errors must be removed before the compiler can successfully recompile the source code again the object program can be executed a number of times without translating it again.

How a Compiler Works:

- Reads the entire source code and breaks it into tokens (keywords, identifiers, operators).
- Checks the tokens against the language's grammatical rules, constructing a parse tree to represent the code structure.
- Validates the logic of the code, ensuring it adheres to the rules (e.g., data type compatibility).
- Improves the performance of the code by optimizing instructions, reducing resource usage, and eliminating redundancies.
- Translates the optimized code into machine language, producing an object code that the CPU can execute.

Interpreter

An **interpreter** is a translator that translates and executes high-level language code line by line, without producing a complete machine code program before execution. Unlike a compiler, which processes the entire code at once, an interpreter translates a single line of code and immediately executes it.

How an Interpreter Works:

- The interpreter reads a single line or instruction from the source code.
- It translates the instruction into machine code and executes it immediately.
- The process continues line by line until the entire program is executed

Comparison Between Assembler, Compiler, and Interpreter

Feature	**Assembler**	**Compiler**	**Interpreter**
Input Language	Assembly Language	High-Level Language	High-Level Language

Output Language	Machine Code (Object Code)	Machine Code (Executable Code)	Executes Directly (No Output)
Execution Time	Fast (already in machine code)	Fast (after compilation)	Slower (line-by-line execution)
Error Detection	Errors detected during assembly	Errors detected before execution	Errors detected at runtime
Code Optimization	Minimal	High	None
Platform Dependency	Dependent on machine architecture	Dependent (machine-specific code)	Platform-independent (as long as interpreter exists)
Examples	NASM, MASM	GCC (C, C++), Java Compiler	Python, JavaScript

Linker

Linker is a utility program in the software development process that plays a critical role in creating a final executable file from object files generated by the compiler or assembler. When a program is written in multiple modules or uses external libraries, the linker is responsible for combining these modules and resolving any references between them. Linkers are also called as link editors.

Role of the Linker

- **Symbol Resolution**: The linker resolves symbols, which are references to variables or functions used in one module but defined in another. If the linker cannot resolve a symbol, it generates an error indicating that the reference is undefined.

- **Relocation**: Each object file assumes it starts at address 0. The linker adjusts these addresses so that all object files can coexist in memory without conflict. This process is called **relocation**, and it ensures that each part of the program is placed in its proper memory location.

- **Library Linking**: The linker must also ensure that external libraries are properly linked to the program. In dynamic linking, this is done by referencing the shared library files, which are loaded at runtime.

- **Generating the Executable File**: Finally, after all modules and libraries are linked, the linker creates the final executable file. This file contains the machine code and is ready for execution by the operating system.

Linking

Linking is a process of collecting and maintaining pieces of code and data into a single file. There are two types of Linking:

1. Static Linking
2. Dynamic Linking

Static Linking

In static linking, all the required libraries and object files are combined into a single executable file during the linking process. The resulting executable contains everything it needs to run independently.

Advantages of Static Linking:

- No need for external libraries at runtime; the program can run on its own.
- Performance may be faster since the program doesn't have to load external libraries at runtime.

Disadvantages of Static Linking:

- The executable file size is larger because it includes copies of the library code.
- Any changes to the library require recompiling and re-linking the program.

Dynamic Linking

In dynamic linking, the program relies on external libraries that are linked at runtime. The program's executable file doesn't include the library code itself but instead includes references to the necessary libraries.

Advantages of Dynamic Linking:

- Smaller executable file size, as libraries are loaded only when needed at runtime.
- Easier to update or modify libraries without needing to recompile the entire program.

Disadvantages of Dynamic Linking:

- The program depends on the availability of the correct versions of the libraries at runtime.
- Slight performance overhead due to loading the libraries during execution.

Testing

Testing is the process of evaluating a software application to ensure that it meets specified requirements, behaves as expected, and is free from defects. It is a critical phase in the software development life cycle (SDLC) because it ensures that the software is reliable, efficient, and meets user needs. Effective testing helps uncover bugs, errors, and inconsistencies that could degrade the quality or functionality of the software.

Software testing can be divided into two steps

Verification: It refers to the set of tasks that ensure that the software correctly implements a specific function. It means “Are we building the product right?”.

Validation: It refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. It means “Are we building the right product?”.

Importance of Testing

Detects Errors: Testing helps identify both obvious and subtle bugs that can cause the software to malfunction or behave unexpectedly.

Ensures Quality: It ensures the software meets the necessary standards, both functionally and non-functionally, providing confidence in its reliability, performance, and security.

Prevents Costly Fixes: Catching and fixing bugs early in the development process is significantly cheaper and faster than doing so after deployment.

Increases Customer Satisfaction: Well-tested software leads to a positive user experience, reducing the chances of user complaints, negative feedback, or system failures in production environments.

Ensures Compliance: In industries like finance or healthcare, rigorous testing ensures the software complies with industry regulations and legal standards.

Types of Testing

There are various types of testing in software development, each designed to focus on specific aspects of the application. These testing types fall under different categories, depending on their purpose and scope.

Functional Testing: Verifies that the software functions according to its requirements. Eg: Unit testing, integration testing, system testing, etc.

Non-Functional Testing: Focuses on aspects such as performance, scalability, usability, and security. Eg: Load testing, stress testing, security testing, and usability testing.

Manual Testing: Performed by human testers who manually execute test cases without the help of automation tools. Best for **exploratory testing**, **usability testing**, and when the software is at an early stage.

Automated Testing: Uses automated tools to run predefined test scripts and compare actual outcomes with expected results. Ideal for **regression testing** and testing large systems with repetitive test cases.

Benefits of Testing:

- Helps identify defects early in the development process.
- Ensures the software is functional, reliable, and usable before deployment.
- Reduces the risk of failure in production environments.
- Helps improve overall customer satisfaction by delivering a quality product.

Debugging

Debugging is the process of identifying, analyzing, and removing bugs or errors in the software. It comes into play after testing has detected issues in the software. Debugging aims to pinpoint the root cause of a defect and correct it so that the software works as intended.

Types of Errors

In programming, errors can be classified into three main types:

1. **Syntax Errors**,
2. **Logical Errors**
3. **Runtime Errors**.

Syntax Errors

Syntax errors occur when the program violates the rules of the programming language. These errors are detected during compilation, and the program won't execute until the errors are fixed.

Characteristics:

- Caused by incorrect use of language syntax.
- Detected by the compiler before the program is executed.
- Common mistakes include missing semicolons, incorrect use of keywords, or unmatched braces.

Example:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!")  // Missing semicolon
    return 0;
}
```

The error here is the missing semicolon (`;`) after the `printf` statement. The correct code is:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!");  // Fixed by adding the semicolon
    return 0;
}
```

Logical Errors

Logical errors occur when the program runs successfully, but the output or behavior is incorrect due to flawed logic. The program executes without crashing, but the results are not as expected. These errors are harder to find because they don't cause a compilation error.

Characteristics:

- Program compiles and runs, but produces incorrect results.
- Caused by wrong logic or incorrect assumptions in the code.
- Not detected by the compiler.

Example:

```c
#include <stdio.h>

int main() {
    int length = 5, width = 10;
    int area = length + width;  // Incorrect logic: should be multiplication
    printf("Area of rectangle: %d\n", area);
    return 0;
}
```

The error here is the logic for calculating the area of a rectangle is incorrect because it uses addition (+) instead of multiplication (*). The corrected code is:

```c
#include <stdio.h>

int main() {
    int length = 5, width = 10;
    int area = length * width;  // Correct logic: multiplication
    printf("Area of rectangle: %d\n", area);
    return 0;
}
```

Runtime Errors

Runtime errors occur while the program is running. These errors are not detected by the compiler but happen due to illegal operations or unexpected conditions during execution, such as dividing by zero or accessing memory that doesn't exist.

Characteristics:

- Occurs during program execution.
- Causes the program to crash or behave unexpectedly.
- Common causes include division by zero, invalid memory access, or file handling errors.

Examples:

Division by Zero:

```c
#include <stdio.h>

int main() {
    int a = 10, b = 0;
    int result = a / b;  // Runtime error: division by zero
    printf("Result: %d\n", result);
    return 0;
}
```

The program crashes because dividing by zero is undefined. You can fix this by checking the divisor:

```c
#include <stdio.h>

int main() {
    int a = 10, b = 0;
    if (b != 0) {
        int result = a / b;
        printf("Result: %d\n", result);
    } else {
        printf("Error: Division by zero is not allowed.\n");
    }
    return 0;
}
```

Summary of Errors:

Error Type	When It Occurs	Detection	Example	Fix
Syntax Error	Before program runs	Compiler	Missing semicolon or unmatched braces	Correct the syntax based on the error message
Logical Error	During execution, gives incorrect results	Not automatically detected	Using + instead of * for area calculation	Review logic and test cases
Runtime Error	During execution	Detected at runtime	Division by zero, invalid array access	Use checks and error handling mechanisms