

GAUHATI UNIVERSITY
Centre for Distance and Online Education

INF-4016

Fourth Semester
(Under CBCS)

M.Sc.- IT

Paper: INF 4016

PROGRAMMING LANGUAGES



CONTENTS:

- Block- I :** **Programming Language Concepts and
Imperative Programming Languages**
- Block- II :** **Object Oriented Languages**
- Block- III :** **Functional Programming Languages and
Logic Programming Languages**

SLM Development Team:

HoD, Department of Computer Science, Gauhati University
Programme Coordinator, M.Sc.-IT, GUCDOE
Prof. Shikhar Kr. Sarma, Department of IT, Gauhati University
Dr. Khurshid Alam Borbora, Assistant Professor, GUCDOE
Dr. Swapnanil Gogoi, Assistant Professor, GUCDOE
Mrs. Pallavi Saikia, Assistant Professor, GUCDOE
Dr. Rita Chakraborty, Assistant Professor, GUCDOE
Mr. Hemanta Kalita, Assistant Professor, GUCDOE

Course Coordination:

Dr. Debahari Talukdar	Director, GUCDOE
Prof. Anjana Kakoti Mahanta	Programme Coordinator, GUCDOE Dept. of Computer Science, G.U.
Dr. Khurshid Alam Borbora	Assistant Professor, GUCDOE
Dr. Swapnanil Gogoi	Assistant Professor, GUCDOE
Mrs. Pallavi Saikia	Assistant Professor, GUCDOE
Dr. Rita Chakraborty	Assistant Professor, GUCDOE
Mr. Hemanta Kalita	Assistant Professor, GUCDOE
Mr. Dipankar Saikia	Editor SLM, GUCDOE

Contributors:

Dr. Khurshid Alam Borbora Assistant Professor, GUCDOE	(Block I : Units- 1 & 2)
Dr. Swapnanil Gogoi Assistant Professor, GUCDOE	(Block II : Units- 1, 2, 3 & 4) (Block III : Unit- 2)
Dr. Pranamika Kakati Assistant Professor Dept. of Computer Science, G.U.	(Block III : Units- 3 & 4)
Mrs. Pallavi Saikia Assistant Professor, GUCDOE	(Block I : Units- 3, 4 & 5)
Mr. Hemanta Kalita Assistant Professor, GUCDOE	(Block III : Unit- 1)

Content Editor:

Dr. Dwipen Laskar	Assistant Professor Dept. of Computer Science, G.U.
--------------------------	--

Cover Page Designing:

Bhaskar Jyoti Goswami	GUCDOE
Nishanta Das	GUCDOE

ISBN: 978-81-986642-9-7

June, 2025

© Copyright by GUCDOE. All rights reserved. No part of this work may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise.
Published on behalf of Gauhati University Centre for Distance and Online Education by the Director, and printed at Gauhati University Press, Guwahati-781014.

CONTENTS:

BLOCK- I : PROGRAMMING LANGUAGE CONCEPTS AND IMPERATIVE PROGRAMMING LANGUAGES

Unit 1: Evolution of Programming Languages	4-19
Unit 2: Programming Methodology-I	20-34
Unit 3: Programming Methodology-II	35-48
Unit 4: Introduction to Imperative Programming Languages	49-66
Unit 5: Concept of Subprogram in Imperative Programming Languages	67-81

BLOCK- II : OBJECT ORIENTED LANGUAGES

Unit 1: Data Abstraction	82-113
Unit 2: Inheritance	114-149
Unit 3: Polymorphism	150-173
Unit 4: Exception Handling	174-196

BLOCK- III : FUNCTIONAL PROGRAMMING LANGUAGES AND LOGIC PROGRAMMING LANGUAGES

Unit 1: Introduction to Functional Programming Languages	197-216
Unit 2: Functional Programming in C++	217-243
Unit 3: Logic Programming Languages-I	244-259
Unit 4: Logic Programming Languages-II	260-291

BLOCK- I

**PROGRAMMING LANGUAGE CONCEPTS AND
IMPERATIVE PROGRAMMING LANGUAGES**

Unit 1: Evolution of Programming Languages

Unit 2: Programming Methodology-I

Unit 3: Programming Methodology-II

Unit 4: Introduction to Imperative Programming Languages

**Unit 5: Concept of Subprogram in Imperative
Programming Languages**

UNIT 1: EVOLUTION OF PROGRAMMING LANGUAGES

Unit Structure:

- 1.1 Introduction
- 1.2 Unit Objectives
- 1.3 Introduction to Programming Languages
 - 1.3.1 Purpose of Programming Languages
 - 1.3.2 Historical Perspective and Current Trends
- 1.4 Factors Influencing the Evolution of Programming Languages
 - 1.4.1 Influence of Computer Architecture
 - 1.4.2 Influence of Operating Systems
 - 1.4.3 Influence of Implementation Methods
- 1.5 Summing up
- 1.6 Answers to Check Your Progress
- 1.7 Possible Questions
- 1.8 References and Suggested Readings

1.1 INTRODUCTION

Programming languages form the foundation of modern computing, enabling human interaction with machines through structured instructions. This unit provides a comprehensive overview of programming languages, their purpose, historical evolution, and the key factors influencing their development. It begins with an introduction to programming languages, covering their syntax, semantics, and pragmatics, followed by a discussion on their role in problem-solving, automation, software development, and artificial intelligence. The historical perspective traces the evolution from machine code and assembly languages to high-level, modular, and object-oriented languages, culminating in modern domain-specific and web-centric languages.

The unit also explores factors influencing the evolution of programming languages, including computer architecture, operating systems, and implementation methods. Advancements in computer architecture, such as parallel processing, memory management, and specialized hardware, have led to new language features and optimizations. Similarly, operating systems have played a crucial role in shaping system calls, networking capabilities, and security mechanisms in programming languages. The implementation methods—compilation, interpretation, Just-In-Time (JIT) compilation, and runtime environments—determine language performance, portability, and ease of use. By understanding these influences, learners will gain insights into why programming languages are designed the way they are and how they continue to evolve with technological advancements.

1.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Understand the fundamental concepts of programming languages, including syntax, semantics, and pragmatics.
- Explain the purpose of programming languages in human-computer interaction, problem-solving, and software development.
- Analyze the historical evolution of programming languages from machine code to modern high-level and domain-specific languages.
- Identify key programming paradigms such as procedural, object-oriented, and functional programming.
- Examine the influence of computer architecture on the design and evolution of programming languages.
- Describe the impact of operating systems on programming languages, including system calls, networking capabilities, and security features.
- Describe the impact of Describe the impact of on programming languages

1.3 INTRODUCTION TO PROGRAMMING LANGUAGES

A programming language is a formal set of instructions that allows humans to communicate with computers and control their behavior. It provides a structured way to write programs, which consist of commands that a computer can interpret and execute.

A programming language consists of:

Syntax: The rules that define the structure of valid statements.

Semantics: The meaning behind those statements.

Pragmatics: How the language is used effectively in real-world applications.

1.3.1 Purpose of Programming Languages

Programming languages serve as a medium through which humans communicate with computers to execute various tasks. They provide a structured way to define computations, automate processes, and develop software applications. Over time, programming languages have evolved to meet the needs of different computing paradigms, including system programming, artificial intelligence, web development, and data science. Let's discuss the purpose of programming languages in detail.

Human-Computer Interaction:

One of the primary purposes of programming languages is to provide a bridge between human logic and machine execution. Computers operate using binary (0s and 1s), which is difficult for humans to interpret and manage. Programming languages introduce higher-level abstractions that allow developers to write instructions in a way that is more readable and understandable.

Early computers were programmed directly using machine code, which was difficult and error-prone. The Assembly Language introduced symbolic representations for instructions but still closely tied to hardware architecture. High-Level Languages provide a more human-friendly way to write programs (e.g., Python, Java, C++). By using programming languages, developers can efficiently control hardware and software without directly dealing with machine code.

Problem-Solving and Algorithm Implementation:

Programming languages are essential for implementing algorithms that solve computational and real-world problems.

Mathematical Computations: Used for numerical analysis, statistical modelling, and scientific computing.

Algorithm Development: Used to implement searching, sorting, and optimization algorithms.

Artificial Intelligence and Machine Learning: Languages like Python, R, and Julia are widely used for AI model training and data analysis.

Abstraction and Automation:

Programming languages abstract complex operations, making it easier to write, debug, and maintain programs. High-level programming languages abstract away low-level details such as memory management, CPU registers, and instruction execution. Also, Programming languages enable automation of repetitive tasks, reducing human effort and minimizing errors. By providing higher levels of abstraction, programming languages simplify software development and improve efficiency.

Portability and Code Reusability:

Programming languages enable developers to write code that can run on different platforms with minimal modification.

Portability: Languages like Java use the "Write Once, Run Anywhere" (WORA) approach, allowing programs to execute on multiple operating systems using the Java Virtual Machine (JVM).

Web applications written in JavaScript can run on any modern web browser without modification.

Code Reusability: Object-oriented programming (OOP) languages like Java, Python, and C++ support modular code design. Libraries and frameworks provide reusable code, reducing development effort.

Software Development:

Programming languages are fundamental to creating software applications across different domains. Programming languages provide developers with tools to build, test, and deploy software solutions efficiently.

- *System Software:* Languages like C and Rust are used to develop operating systems, compilers, and firmware.
- *Application Software:* Java, Python, and C++ are widely used for desktop applications.
- *Web Development:* JavaScript, HTML, CSS, and backend languages like PHP and Python enable the development of dynamic websites.
- *Mobile Development:* Swift (for iOS) and Kotlin (for Android) allow mobile app development.
- *Embedded Systems:* C and Assembly are commonly used for programming microcontrollers and embedded devices.

Scalability and Performance Optimization:

Programming languages play a crucial role in building scalable and high-performance applications. C and Fortran are used for applications that require extreme computational power, such as simulations and scientific computing. Languages like Java, Go, and Python are optimized for cloud-based and distributed systems. Languages like Erlang, Rust, and Java provide features for handling multiple processes and threads efficiently. By selecting appropriate programming languages, developers can ensure their applications perform efficiently under different workloads.

Enabling Artificial Intelligence and Data Processing:

Modern programming languages are heavily used in artificial intelligence, machine learning, and big data applications.

Python and R are widely used in AI due to their rich ecosystem of libraries (e.g., TensorFlow, PyTorch, Scikit-learn), e.g., Image recognition, natural language processing (NLP). SQL, Python (Pandas), and Scala (Apache Spark) help process and analyze large datasets. E.g., Real-time analytics for recommendation systems (Netflix, Amazon). Programming languages empower businesses and researchers to leverage data for decision-making and innovation.

Programming languages are essential tools for modern computing, enabling developers to write software that powers businesses, science, and everyday life.

1.3.2 Historical Perspective and Current Trends

Programming languages have changed a lot over time. This change has been driven by new technology, the increasing complexity of computing tasks, and the need to make programming easier and more efficient. The history of programming languages can be divided into different stages, starting from basic machine code to the modern high-level and specialized languages we use today.

Machine code consists of binary instructions (0s and 1s) directly executed by a computer's processor. These codes are Specific to each processor (non-portable), Difficult to read, write, and debug and Prone to errors and time-consuming to program. Machine codes were used for early computers like the ENIAC (1945) and UNIVAC (1951). A deep knowledge of hardware architecture is required for this kind of coding.

Assembly language introduced mnemonic symbols to represent machine instructions, making programming slightly more human-readable. The characteristics of assembly language are:

- One-to-one correspondence with machine code instructions.
- Uses symbols like MOV, ADD, and SUB instead of binary codes.
- Still architecture-dependent and requires an assembler to translate into machine code.

Early systems like IBM 360, PDP-11 used this language. Compared to machine code, assembly language had improved readability and debugging but still required knowledge of hardware details.

After assemble language, High-level Programming Languages introduced english-like syntax and abstraction from hardware, making programming easier and more portable. This kind of language is closer to human language, abstracts hardware details and requires compilers or interpreters to translate into machine code. The Major developments are – FORTRAN, COBOL, LISP, ALGOL etc. This kind of programming language enabled complex problem-solving with easier syntax, increased portability across different computers and reduced programming errors and development time.

Next, the era of modular programming started. Modular programming introduced block-structured languages. It emphasized

on code readability and maintainability. It has better error handling and debugging capabilities. The major developments are – C, Pascal, Prolog, Ada etc. C became the foundation for modern operating systems (Linux, Windows). The structured programming improved maintainability and reusability. And thus stronger data abstraction principles influenced later languages.

The Object-oriented programming (OOP) introduced concepts like encapsulation, inheritance, and polymorphism, enabling better software organization and reuse. It uses objects and classes to model real-world entities. It encourages code reuse through inheritance and also supports modular programming for large applications. C++, Smalltalk, Java, Python are the major developments. Java revolutionized web development. C++ became widely used for game development and systems programming. Python gained popularity in AI, data science, and automation.

The rise of the internet led to scripting languages designed for web development and automation. These are designed for rapid development and ease of use. They often used for web and server-side scripting. JavaScript enabled dynamic content in web browsers; PHP used for server-side scripting in web applications; Ruby focused on developer happiness (Ruby on Rails framework) etc. Thus scripting languages enabled interactive web applications, simplified web and backend development and thus paved the way for modern web technologies (Node.js, React, Angular).

Modern and domain-specific Languages focus on security, concurrency, functional programming, and domain-specific applications. The major characteristics of those kind of languages are - Improved performance, safety, and concurrency; Emphasis on cloud computing, AI, and data science; Increased use of functional programming paradigms etc.

Programming languages have evolved to make coding easier, more efficient, and more secure. From simple machine code to powerful high-level languages, each step has made software development more accessible, flexible, and productive. Today, programming languages continue to improve, helping developers create better and more advanced software for the future.

1.4 FACTORS INFLUENCING THE EVOLUTION OF PROGRAMMING LANGUAGES

The evolution of programming languages has been shaped by various technological, computational, and human factors. Among them, computer architecture, operating systems, and implementation methods play crucial roles. These elements define the capabilities, performance, and usability of programming languages, influencing their development over time.

1.4.1 Influence of Computer Architecture

Computer architecture refers to the structure and organization of a computer system, including its hardware components, instruction set, and data processing capabilities. As hardware evolves, programming languages adapt to take advantage of new capabilities.

The design and evolution of programming languages have been significantly influenced by computer architecture. One key factor is the Instruction Set Architecture (ISA), which determines how a processor executes instructions. Different ISAs, such as CISC (Complex Instruction Set Computing) and RISC (Reduced Instruction Set Computing), require different language design approaches. Assembly languages were tightly linked to specific ISAs (e.g., x86 Assembly vs. ARM Assembly), while high-level languages abstract these differences, though performance optimizations still consider underlying architecture. Another major influence is the shift toward Parallel and Multicore Architectures.

Early languages like FORTRAN and COBOL were designed for sequential execution, but the rise of multicore processors led to the development of concurrency-focused languages like Go, Rust, and Erlang. Even widely used languages like Java and Python evolved to include threading and parallel computing models to better utilize multiple CPU cores. Memory Management and Addressing have also played a crucial role in language development. Older systems required manual memory management using functions like malloc and free in C, whereas modern languages such as Java and Python

introduced automatic garbage collection to simplify memory handling. Additionally, advancements in virtual memory and cache hierarchies have influenced memory-efficient programming techniques.

Finally, the rise of Specialized Hardware, including GPUs and AI accelerators, has led to the creation of domain-specific languages. For instance, CUDA was developed for NVIDIA GPUs, while OpenCL enables parallel processing across different hardware. Similarly, AI and machine learning growth have led to specialized languages like TensorFlow and Julia, which cater to scientific computing and data-intensive applications. These architectural influences continue to shape modern programming languages, ensuring they align with the evolving hardware landscape.

1.4.2 Influence of Operating Systems

Operating systems (OS) provide an interface between hardware and software, influencing programming paradigms, system calls, and runtime behavior.

The evolution of programming languages has also been heavily influenced by operating system (OS) design and functionality. System Calls and APIs played a crucial role in shaping early languages like Assembly and C, which were closely tied to low-level OS functions such as file handling and process control. As computing advanced, higher-level languages emerged to abstract OS dependencies, with Java's JVM enabling cross-platform execution.

Modern languages like Swift (for iOS) and Kotlin (for Android) are deeply integrated with mobile OS environments, optimizing application development for specific platforms. Another key influence is Multi-User and Networking Capabilities. The development of UNIX in the 1970s influenced C's process control

and networking APIs, leading to more sophisticated networking-oriented languages like Erlang, designed for distributed telecom systems.

The rise of the internet further drove the evolution of web-based languages like JavaScript and PHP, which are essential for modern web applications. Security and Access Control have also shaped programming language features. OS security requirements led to the development of sandboxing techniques in Java and JavaScript, while modern system programming languages like Rust enforce memory safety through strict ownership and borrowing rules. These advancements help prevent vulnerabilities such as memory leaks and buffer overflows.

Lastly, Real-Time and Embedded Systems require predictable execution times, influencing the creation of languages like Ada, widely used in critical applications like avionics and defense. C and Assembly remain dominant in embedded systems due to their efficiency and low overhead, ensuring precise control over hardware resources. As operating systems continue to evolve, programming languages will adapt to enhance performance, security, and platform compatibility.

1.4.3 Influence of Implementation Methods

The implementation method of a programming language has a major impact on its performance, portability, and ease of use. One of the key distinctions in implementation is Compilation vs. Interpretation.

Compiled languages like C, C++, and Rust convert source code into machine code before execution, resulting in high performance and efficiency. However, they require recompilation for different platforms. On the other hand, Interpreted languages like Python, JavaScript, and Ruby translate and execute code line by line at

runtime. This makes debugging easier and enhances cross-platform compatibility, but at the cost of slower execution speed. Some languages use Hybrid Approaches, such as Java, which compiles code into bytecode that runs on the Java Virtual Machine (JVM), balancing performance and portability. Similarly, Python's CPython is an interpreter, but it also supports compilation methods like PyPy and Cython. Another optimization method is Just-In-Time (JIT) Compilation, used in Java (JVM), JavaScript (V8 engine), and .NET (CLR), which translates bytecode into machine code at runtime to improve execution speed while maintaining portability.

Virtual Machines and Runtime Environments also play a critical role in language portability. The JVM allows Java programs to run across different OS platforms, while .NET CLR enables interoperability among multiple languages like C#, F#, and VB.NET. Additionally, WebAssembly (Wasm) provides near-native performance for languages like Rust, C++, and Python in web environments.

Memory management strategies also influence language design. Early languages like C required manual memory management, while automatic garbage collection in languages like Java, Python, and C# improved memory safety and developer productivity. More recently, memory safety mechanisms in languages like Rust and Swift reduce the need for garbage collection while ensuring security.

Another major design factor is Dynamic vs. Static Typing. Statically typed languages such as C, Java, and Rust perform type checking at compile time, improving performance and reducing runtime errors. Dynamically typed languages like Python, JavaScript, and Ruby offer greater flexibility but can introduce errors during execution. Some modern languages like TypeScript and Kotlin offer optional static typing, striking a balance between flexibility and safety.

Overall, the choice of implementation method shapes a language's efficiency, security, and ease of development, influencing how it is used in various computing environments.

CHECK YOUR PROGRESS-I

1. State True or False:

- a) Programming languages provide a structured way to define computations and automate processes.
- b) Machine code is highly portable and easy to debug.
- c) Assembly language introduced symbolic representations for machine instructions, making programming more human-readable.
- d) Scripting languages like JavaScript and PHP are primarily designed for system programming.

2. Fill in the Blanks:

- a) The two main components of a programming language are _____ and _____.
- b) The primary role of an operating system is to provide an interface between hardware *and* _____.
- c) A programming paradigm that focuses on breaking programs into small, reusable functions is called _____.
- d) The instruction set architecture (ISA) determines how a processor executes _____.
- e) The Just-In-Time (JIT) compilation technique improves performance by translating _____.

1.5 SUMMING UP

A programming language is a formal way for humans to communicate with computers, defining how instructions are written

and executed. It consists of syntax (rules for valid statements), semantics (meaning of statements), and pragmatics (real-world usage). Programming languages have evolved to improve human-computer interaction, making it easier to control hardware and automate processes. They facilitate problem-solving, algorithm implementation, and abstraction, reducing complexity and enhancing productivity.

Key benefits include portability (e.g., Java's "Write Once, Run Anywhere"), code reusability (via object-oriented programming), and support for software development across multiple domains, including system software, web, mobile, and embedded systems. Performance optimization and scalability are also critical, with languages like C, Java, and Python used for high-performance and distributed computing. Additionally, modern programming languages play a significant role in AI, machine learning, and data processing, enabling advanced analytics and automation.

The factors Influencing Programming Language Evolution are:

- **Computer Architecture:** Changes in instruction sets, parallel computing, and memory management have shaped language design. The shift to multicore processing led to concurrency-focused languages like Go and Rust, while specialized hardware (e.g., GPUs) inspired domain-specific languages like CUDA and OpenCL.
- **Operating Systems:** OS influence programming via system calls, networking capabilities, and security features. UNIX shaped C's process control, while Java's JVM enabled cross-platform execution. Mobile OS environments drove the adoption of Swift (iOS) and Kotlin (Android). Security concerns led to memory-safe languages like Rust.

- **Implementation Methods:** The choice between compiled (C, Rust) and interpreted (Python, JavaScript) languages affects performance and portability. Hybrid approaches, such as Java's bytecode (JVM) and JIT compilation, balance efficiency and flexibility. Modern runtime environments like Web Assembly extend language capabilities to the web.

1.6 ANSWERS TO CHECK YOUR PROGRESS

- 1.a) True b) False c) True d) False
- 2.a) Syntax, Semantics b) Software c) Modular programming
d) Instruction e) Bytecode into machine code at runtime

1.7 POSSIBLE QUESTIONS

Short Answer Type Questions:

1. What are the three main components of a programming language?
2. Why is portability an important factor in programming languages?
3. How did object-oriented programming improve software development?
4. What is the role of scripting languages in web development?
5. What are some key influences of computer architecture on programming languages?

Long Answer Type Questions:

6. Discuss the impact of object-oriented programming (OOP) on modern software development, including key concepts.
7. Explain the influence of operating systems on programming languages, focusing on system calls, networking, and security.
8. Compare and contrast compilation and interpretation, explaining how they impact programming language performance.

9. How have modern trends in programming languages, such as security, concurrency, and domain-specific languages, shaped software development today?

1.8 REFERENCES AND SUGGESTED READINGS

1. Sebesta, Robert W. *Concepts of programming languages*. Pearson Education India, 2016.

---X---

UNIT 2: PROGRAMMING METHODOLOGY I

Unit Structure:

- 2.1 Introduction
- 2.2 Unit Objectives
- 2.3 Introduction to Programming Methodology
- 2.4 Development in Programming Methodology
- 2.5 Desirable Features in Programming Languages
- 2.6 Design Issues in Programming Methodology
- 2.7 Summing up
- 2.8 Answers to Check Your Progress
- 2.9 Possible Questions
- 2.10 References and Suggested Readings

2.1 INTRODUCTION

The development of programming methodology refers to the evolution of techniques and principles used in writing efficient, maintainable, and scalable software. Initially, programming was done in machine and assembly languages, which were complex and hardware-dependent. With time, structured programming introduced modularity and readability, followed by object-oriented programming (OOP), which emphasized code reuse and abstraction. Functional and declarative paradigms further expanded programming capabilities. Modern methodologies integrate multiple paradigms, concurrency, and domain-specific approaches to enhance efficiency and adaptability in software development.

2.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Understand the Concept of Programming Methodology.
- Recognize the Benefits of Systematic Programming Approaches.
- Describe the historical development of programming paradigms including procedural, object-oriented, functional, event-driven, and concurrent programming.
- Evaluate Desirable Features in Programming Languages.

- Analyze Design Issues in Programming Methodology.

2.3 INTRODUCTION TO PROGRAMMING METHODOLOGY

Programming methodology refers to the systematic approach used in designing, writing, testing, and maintaining software. It defines best practices, principles, and techniques that help developers create efficient, reliable, and maintainable programs.

In the early days, programming was done using machine code and assembly language, making development error-prone and hardware-dependent. Over time, structured programming introduced better readability and organization, followed by object-oriented programming (OOP), which emphasized modularity and reuse. Later, functional and declarative paradigms emerged to enhance problem-solving approaches.

Modern programming methodologies integrate various paradigms, emphasize agile development, and focus on software quality, scalability, and security. The evolution of programming methodology ensures that software development keeps pace with technological advancements and growing computational demands. Below are the key reasons why programming methodology is important:

Improves Code Readability and Maintainability: A well-defined programming methodology promotes writing clean and well-structured code, ensuring better readability and maintainability. By following proper naming conventions, indentation, and comments, developers can make the code more understandable, reducing ambiguity and confusion. This structured approach facilitates team collaboration, allowing multiple developers to work on the same codebase efficiently. For example, using meaningful variable names enhances clarity, while writing modular code with functions or classes improves reusability, making the development process more efficient and scalable.

Enhances Software Reliability and Debugging: Following systematic programming practices reduces the chances of errors and bugs, ensuring that software functions as intended. Implementing error handling mechanisms, such as exception handling in Java and Python, helps identify and manage runtime errors efficiently,

preventing unexpected crashes and improving program stability. Additionally, proper testing methodologies, including unit testing, integration testing, and system testing, verify that the software behaves as expected in different scenarios. By adopting these practices, developers can enhance software reliability, minimize debugging efforts, and deliver high-quality applications.

Increases Development Efficiency and Productivity:By following a structured methodology, developers can focus on solving problems rather than dealing with unnecessary complexities, making the software development process more efficient. Code reuse through functions, libraries, and frameworks significantly speeds up development by reducing redundancy and allowing developers to leverage pre-existing solutions. Agile methodologies and modular programming further enhance productivity by enabling teams to work in parallel on different parts of a project, ensuring faster iterations and continuous improvements. For example, using Python's built-in libraries for data analysis eliminates the need to write custom algorithms from scratch, saving time and effort. Similarly, adopting agile development cycles facilitates iterative enhancements, allowing developers to refine and optimize software based on feedback and evolving requirements.

Ensures Scalability and Performance Optimization:Good programming practices lead to efficient code that runs faster and consumes fewer resources, making software more responsive and scalable. Optimized data structures and algorithms play a crucial role in enhancing performance, especially in large-scale applications where processing vast amounts of data efficiently is essential. Scalable design principles ensure that software can handle increased workloads, adapting to growing data and user demands without compromising speed or reliability. For example, using hash tables for fast lookups significantly improves search efficiency compared to linear searches in large datasets. Additionally, implementing multi-threading allows applications to leverage multiple CPU cores, enabling parallel execution and improving overall system performance. By incorporating these best practices, developers can create high-performance, scalable, and resource-efficient software solutions.

Supports Code Reusability and Modularity:Writing modular code allows developers to reuse components across different

projects, improving efficiency and reducing redundancy. By breaking down software into smaller, self-contained modules, developers can easily integrate and modify existing code without rewriting it. Object-oriented programming (OOP) further enhances reusability through principles such as inheritance, which allows new classes to derive properties and behaviors from existing ones, and polymorphism, which enables a single interface to represent different underlying data types. Additionally, functional programming promotes reusability by emphasizing pure functions—which produce the same output for given inputs without side effects—and higher-order functions, which take other functions as arguments or return them as results. These methodologies collectively contribute to writing scalable, maintainable, and reusable code, ultimately improving software development productivity.

Enhances Security and Robustness: Secure programming practices are essential in preventing vulnerabilities such as SQL injection, buffer overflow, and cross-site scripting, which can compromise software security. By adopting sandboxing techniques and memory safety mechanisms, developers can minimize security risks and ensure that applications run in a controlled environment. Modern programming languages like Rust and Swift incorporate built-in memory safety features, such as ownership models and automatic memory management, to prevent common security pitfalls like dangling pointers and memory leaks. These proactive measures enhance software reliability and protect systems from potential cyber threats.

2.4 DEVELOPMENT IN PROGRAMMING METHODOLOGY

The evolution of programming approaches has been driven by continuous advancements in hardware, software requirements, and the increasing complexity of computing tasks. Over time, programming methodologies have transformed to enhance efficiency, maintainability, and scalability in software development.

The major programming paradigms can be categorized into different stages, each addressing specific challenges in programming. The earliest approach was machine code, where instructions were directly written in binary (0s and 1s). This was complex, time-

consuming, and prone to errors. To improve efficiency, assembly language introduced mnemonic symbols such as MOV, ADD, and SUB, making programming slightly easier but still closely tied to hardware.

As software systems grew, the need for structured and modular code led to procedural programming, which introduced structured programming techniques. Procedural programming used functions to break down tasks, followed a top-down approach, and introduced control flow structures like loops and conditionals. Languages like FORTRAN, COBOL, and C enabled modular programming and reduced reliance on low-level machine instructions.

The procedural programming had limitations in managing large-scale applications, which led to the rise of object-oriented programming (OOP). OOP models real-world entities using objects and classes, incorporating concepts like encapsulation, inheritance, and polymorphism to improve code reuse and modularity. Popular OOP languages include C++, Java, and Python, widely used in enterprise applications and graphical user interfaces. An alternative approach, functional programming, treats computation as mathematical functions, emphasizing immutability and pure functions. It supports parallel processing and is widely used in data analysis and distributed computing. Languages like Lisp, Haskell, and Scala follow this paradigm.

The rise of graphical interfaces and web applications led to event-driven and reactive programming, where programs respond to user interactions or system events using event listeners and callbacks. JavaScript, Node.js, and ReactiveX frameworks are widely used in UI development and real-time applications.

As computing moved towards multicore and distributed environments, concurrent and parallel programming emerged, enabling efficient task execution across multiple processors. Languages like Go, Rust etc. facilitate high-performance computing, AI, and cloud-based applications.

In modern computing, domain-specific languages (DSLs) have been developed for specialized tasks, providing optimized syntax and libraries for fields such as databases (SQL), web development (HTML/CSS), and machine learning (TensorFlow, PyTorch). The evolution of programming approaches has significantly improved software development by enhancing maintainability, security, and

scalability. Today, modern programming blends multiple paradigms, using object-oriented, functional, event-driven, and concurrent programming together to create robust and efficient applications. As technology advances, programming methodologies will continue to evolve, adapting to new computational challenges.

2.5 DESIRABLE FEATURES IN PROGRAMMING LANGUAGES

A programming language serves as a bridge between human logic and machine execution, allowing developers to create efficient and reliable software. To facilitate effective programming, a language must possess certain desirable features that enhance its usability, performance, and security. These features influence a language's adoption, efficiency, and ease of maintenance. Below are some of the most important desirable features in programming languages.

Simplicity and Readability: A programming language should have a simple and clear syntax that is easy to understand and use. Simplicity reduces the learning curve for new developers and makes the code easier to maintain. Languages with minimal complexity avoid unnecessary syntax and constructs that can make coding difficult. Additionally, a language should allow for the natural expression of logic, resembling mathematical notation or natural language, to enhance readability. A consistent and intuitive syntax, like that of Python, improves code clarity by emphasizing indentation and structure, making programs easier to follow and debug.

Expressiveness and Conciseness: A language should allow programmers to express ideas efficiently with minimal code. High-level languages like Python and JavaScript offer rich built-in functions and extensive libraries that accelerate development. Code reusability is another important factor—object-oriented languages like Java and C++ promote reusability through classes and inheritance. Furthermore, a programming language should provide higher-level abstraction, enabling developers to focus on solving problems rather than dealing with low-level system details.

Performance and Efficiency: Programming languages should execute efficiently in terms of speed and resource utilization. Optimized compilation or interpretation plays a key role in

performance. Compiled languages such as C and Rust translate code directly into machine instructions for faster execution. Efficient memory management, including garbage collection in Java and automatic memory safety in Rust, helps in improving performance by preventing memory leaks. Additionally, languages designed for high-performance computing, such as C and Rust, ensure minimal overhead, making them suitable for embedded systems and real-time applications.

Portability and Platform Independence: A language should allow programs to run on different platforms with minimal modifications. Cross-platform support is crucial for modern applications, as it ensures software can run on multiple operating systems. Java achieves this through its "Write Once, Run Anywhere" (WORA) approach, which relies on the Java Virtual Machine (JVM). Standardized libraries and APIs further enhance portability by providing consistent functionality across different platforms. Moreover, support for virtual machines, such as WebAssembly (Wasm), allows languages like C++ and Rust to run efficiently across diverse environments.

Robustness and Reliability: A programming language should minimize errors and handle exceptions efficiently to ensure stable software. A strong typing system is one of the key factors in reliability—statically typed languages like Java, C#, and Rust enforce type checks at compile time, reducing runtime errors. Additionally, modern programming languages provide exception handling mechanisms, such as try-catch blocks, to manage errors effectively. Memory safety is another critical aspect of robustness; languages like Rust use ownership models, and Swift provides automatic memory management to prevent common security vulnerabilities like buffer overflows and memory leaks.

Security Features: Security is a fundamental concern in software development, particularly in web applications and system programming. A secure programming language should enforce memory safety, as seen in Rust and Swift, to prevent vulnerabilities such as buffer overflow attacks. Sandboxing is another important feature—it isolates code execution from the operating system, ensuring malicious code does not compromise system security. Additionally, modern languages provide built-in security libraries that offer encryption, authentication, and access control mechanisms, helping developers build secure applications.

Scalability and Modularity: A language should support the development of scalable applications that can handle increasing workloads efficiently. Modular programming support is essential for scalability, allowing developers to write reusable and maintainable code. Object-oriented programming (OOP) and functional programming paradigms facilitate modular design. Parallel processing capabilities, as seen in languages like Go and Erlang, enable applications to scale effectively by utilizing concurrency. Moreover, strong framework support, such as Django for Python and Spring for Java, helps developers build large-scale applications efficiently.

Support for Multiple Paradigms: Modern programming languages should support multiple programming paradigms to provide flexibility in software development. Procedural programming, found in languages like C and Pascal, allows structured programming with functions and loops. Object-oriented programming (OOP), used in Java, C++, and Python, enables better code organization through classes and objects. Additionally, functional programming, supported in languages like Haskell, Lisp, and Scala, promotes immutability and higher-order functions, which enhance modularity and reduce side effects in code execution.

Dynamic and Static Typing: Typing systems affect how variables and data are handled in a programming language. Statically typed languages such as C, Java, and Rust enforce type constraints at compile time, reducing errors and improving performance. On the other hand, dynamically typed languages like Python and JavaScript do not require explicit type declarations, making development faster but increasing the likelihood of runtime errors. Hybrid typing systems, such as those in TypeScript and Kotlin, allow optional static typing to balance flexibility with type safety.

Maintainability and Debugging Support: A programming language should facilitate easy debugging, testing, and long-term maintenance of code. Readability plays a significant role in maintainability—languages with well-structured syntax and clear coding conventions make it easier to understand and modify code. Integrated debugging tools, such as those available in Python and Java, assist developers in identifying and fixing errors efficiently. Furthermore, compatibility with version control systems like Git ensures that code changes can be tracked and managed effectively, making collaboration and maintenance easier in large-scale projects.

The choice of a programming language depends on its desirable features, which determine how efficiently developers can write, debug, and maintain code. While some languages prioritize performance (e.g., C, Rust), others focus on ease of use and rapid development (e.g., Python, JavaScript). A well-designed programming language should strike a balance between performance, security, scalability, and usability, enabling developers to create robust and efficient software solutions.

2.6 DESIGN ISSUES IN PROGRAMMING METHODOLOGY

The design of a programming language is influenced by several factors, including usability, efficiency, reliability, and security. These design issues play a crucial role in determining how a language is structured, how it executes code, and how developers interact with it. Understanding these factors helps in developing robust and efficient programming languages suited to various applications.

- **Syntax and Readability**

The syntax of a language defines how statements are written and structured. Readability is crucial because well-structured and intuitive syntax makes the language easier to learn and use. Some languages, like Python, prioritize simple and clean syntax, while others, such as Perl, allow more flexibility but may lead to complex and unreadable code.

Key considerations for syntax and readability include whether the language should use keywords (e.g., `if`, `else`, `while`) or symbolic representations (e.g., `{}`, `;`). Another important aspect is whether indentation and whitespace should be significant, as in Python, or ignored, as in C and Java.

- **Type System and Type Safety**

The type system determines how variables and functions handle data. Strongly typed languages, such as Java and Rust, enforce strict type rules, reducing runtime errors. In contrast, weakly typed languages like JavaScript and Python offer flexibility but can lead to unexpected behavior.

Key considerations include static vs. dynamic typing, where types can be checked at compile-time (e.g., C, Java) or at runtime (e.g.,

Python, JavaScript). Another design choice is explicit vs. implicit typing—should the developer declare types explicitly, as in C and Java, or should the language infer them automatically, as seen in Swift, Kotlin, and TypeScript?

- **Memory Management**

Memory management is critical for performance and security. Manual memory management, as seen in C with malloc and free, offers control but increases the risk of memory leaks and segmentation faults. On the other hand, automatic memory management via garbage collection, as in Java and Python, simplifies development but may introduce performance overhead.

Languages must decide whether to rely on garbage collection or require manual memory handling. Some modern languages, like Rust, enforce memory safety through ownership models, eliminating memory leaks and buffer overflows.

- **Concurrency and Parallelism**

With the rise of multicore processors, efficient handling of multiple tasks simultaneously is essential. Some languages, such as Go, Rust, and Erlang, are designed with built-in concurrency support, while others, like Python, require additional libraries or frameworks.

Key considerations include whether the language should support multithreading and parallel execution natively. Another aspect is the communication model between threads—should the language use shared memory (as in Java and C++) or message passing (as in Erlang and Go)?

- **Security Features**

Security is a major concern in programming language design. Some languages, such as Rust, enforce memory safety, preventing issues like buffer overflows. Others, like Java and JavaScript, implement sandboxing techniques to isolate code execution and reduce risks.

Design considerations include whether the language should include features to prevent common vulnerabilities like SQL injection and buffer overflow. Another key aspect is enforcing strict access control and sandboxing mechanisms to improve security.

- **Portability and Platform Independence**

Portability allows a language to run on different platforms with minimal changes. Java achieves this through the Java Virtual Machine (JVM), while C and C++ rely on platform-specific compilation. Web-based languages like JavaScript and Python are inherently cross-platform.

Languages must decide whether to be compiled for each platform or use a virtual machine. Another consideration is whether they should support multiple architectures natively to enhance their applicability across different environments.

- **Error Handling and Exception Management**

Proper error handling improves software robustness and maintainability. Languages like Java and Python use structured exception handling (e.g., try-catch blocks), while C relies on error codes, making error handling more complex.

Key considerations include whether the language should support built-in exception handling mechanisms and whether errors should be checked at compile-time or runtime to ensure reliability.

- **Paradigm Support (Imperative, Functional, Object-Oriented, etc.)**

Programming paradigms define the approach to solving problems. Some languages, such as Java and C++, focus on object-oriented programming, while others, like Haskell and Lisp, emphasize functional programming. Modern languages like Python and Kotlin support multiple paradigms.

A key design decision is whether the language should support only one paradigm (e.g., C, Java) or multiple paradigms (e.g., Python, Scala). Another factor is whether the language should allow features like first-class functions and immutability, which are central to functional programming.

- **Standard Library and Ecosystem**

A rich standard library simplifies development by providing built-in functionality. Python, for instance, has extensive libraries for AI and data science, while JavaScript has frameworks for web development.

Languages must decide whether to provide a minimal standard library (e.g., C) or an extensive one (e.g., Python). Additionally, built-in support for networking, data processing, and concurrency enhances a language's usability and versatility.

- **Compilation vs. Interpretation**

Compiled languages, such as C and Rust, translate code into machine code before execution, improving performance. Interpreted languages, such as Python and JavaScript, execute code line-by-line, offering flexibility but reducing speed. Some languages, like Java, use a hybrid approach with Just-In-Time (JIT) compilation.

Key considerations include whether the language should be compiled, interpreted, or use a hybrid model. Additionally, designers must balance execution speed with flexibility to meet different application needs.

The design of a programming language is influenced by various factors, including readability, memory management, security, portability, and execution efficiency. Each design choice impacts how developers interact with the language and how applications perform in real-world scenarios. By addressing these design issues effectively, programming languages can enhance usability, reliability, and performance, making them suitable for diverse computing environments.

CHECK YOUR PROGRESS-I

1. State True or False:

- a) Programming methodology defines best practices and techniques for writing maintainable software.
- b) Object-Oriented Programming enhances code reuse through inheritance and polymorphism.
- c) Machine code is easy to write and maintain.
- d) Memory safety features in languages like Rust help prevent buffer overflows and memory leaks.
- e) Procedural programming is more suitable than object-oriented programming for large-scale applications.

2. Fill in the Blanks:

- a) Programming methodology refers to the _____ used in designing, writing, testing, and maintaining software.
- b) _____ was introduced to improve code readability and organization.
- c) In functional programming, _____ produce the same output for given inputs without side effects.
- d) _____ involves techniques like garbage collection or ownership models to handle resource allocation.
- e) Python emphasizes syntax clarity and uses _____ as a significant part of code structure.

2.7 SUMMING UP

- Programming methodology refers to a structured approach to software development, emphasizing best practices in design, implementation, testing, and maintenance.
- Key benefits include improved code readability, reduced errors through systematic debugging and testing, enhanced development productivity via reusable components and agile

practices, and support for scalable and high-performance applications.

- Programming methodology also reinforces software robustness and security by incorporating safe memory practices and preventive measures against vulnerabilities.
- Programming methodologies have developed alongside advances in hardware and software complexity. Early approaches used machine and assembly code, which were error-prone and hardware-dependent.
- Object-oriented programming (OOP) followed, supporting real-world modeling, inheritance, and encapsulation. Functional programming introduced pure functions and immutability, ideal for data analysis and concurrency.
- The growth of interactive applications gave rise to event-driven and reactive programming, while parallel and concurrent programming became vital in multicore and cloud environments.
- Modern development blends multiple paradigms and employs domain-specific languages to meet specialized needs, ensuring scalability, maintainability, and security.
- An effective programming language must be simple, readable, expressive, and concise, allowing developers to write clear and maintainable code.
- Programming language design is influenced by various factors. Syntax and readability impact ease of learning, while the type system affects safety and flexibility.
- Languages must also address concurrency, parallelism, and error handling mechanisms for modern computing needs. Security features such as buffer overflow protection and sandboxing are critical.
- Portability ensures cross-platform compatibility, and paradigm support (imperative, functional, OOP) increases versatility.
- A rich standard library improves development efficiency, and decisions around compilation versus interpretation affect performance and flexibility.

2.8 ANSWERS TO CHECK YOUR PROGRESS

- 1.a) True b) True c) False d) True e) False
- 2.a) Systematic approach b) Structured programming
c) Pure functions d) Memory management e) Indentation

2.9 POSSIBLE QUESTIONS

Short Answer Type Questions:

1. Mention any two key reasons why programming methodology is important.
2. Name two programming paradigms introduced after object-oriented programming.
3. Define the concept of memory safety in programming languages.
4. What is meant by platform independence in programming languages?
5. Mention any two desirable features of a programming language.

Long Answer Type Questions:

6. Discuss the importance of programming methodology in software development. Provide examples.
7. Explain how programming methodology has evolved from machine code to modern paradigms.
8. Discuss various programming paradigms and their impact on software design and development.
9. Compare compiled and interpreted languages in terms of performance and flexibility.
10. What is concurrency in programming? How is it supported in modern programming languages?

2.10 REFERENCES AND SUGGESTED READINGS

- Sebesta, Robert W. *Concepts of programming languages*. Pearson Education India, 2016.

---X---

UNIT 3: PROGRAMMING METHODOLOGY II

Unit Structure:

- 3.1 Introduction
- 3.2 Objectives
- 3.3 Language Processors
- 3.4 Syntax, Semantics and Virtual Machines
- 3.5 Binding and Binding Time
- 3.6 Summing Up
- 3.7 Answers to Check Your Progress
- 3.8 Possible Questions
- 3.9 References and Suggested Readings

3.1 INTRODUCTION

The software tools that helps a computer to understand and execute what is required by translating high level language program into low level language is called a language processor. Based on the programming language, there are different types of language processors like assembler, interpreter and compiler. With the help of language processors, programmers can write codes easily and make it functional on computer systems.

3.2 OBJECTIVES

After going through this unit, you will be able to

- Understand different types of language processors
- Understand the syntax and semantics of programming language
- Understand the virtual machine
- Understand binding and the binding time
- Understand the types of binding

3.3 LANGUAGE PROCESSORS

A language processor is a computer program that translates source code from one programming language into other language. They also detect the errors occurred during the translation. All computer programs are written in high level languages like C, C++, Python, Java. To make this language understandable by the computer, it is translated into machine codes. The machine codes are also known as object codes and is made up of only ones and zeros. There are different types of language processors: assembler, interpreter and compiler.

- **Assembler:** Assembler is used to translate a program written in assembly language into machine language. The input to an assembler is a source code that consists of assembly language instructions. Assembly language is the first language that provided an interface for interaction between human being and a computer. Assembly language use certain codes called mnemonics to carry out the different tasks. Some examples of mnemonics are ADD, SUB, MUX, DIV and so on. These mnemonics are converted to binary or the machine codes by the computer. Every computer have it's own set of instructions, as mnemonics depend upon the architecture of the computer.

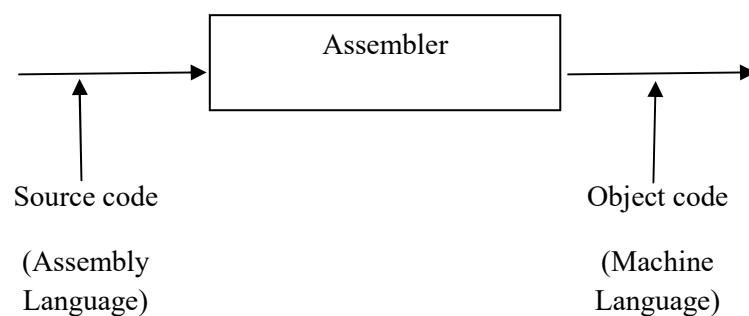


Fig 1: Assembler

- **Interpreter:** Interpreter translates a single line in the source program into machine code and executes it immediately before proceeding towards the next line. The interpreter moves to the next line only after checking whether there is

any error and if any error is present, it corrects it. Programming languages like Python, Ruby and Javascript uses interpreter. As interpreter translates each statement line by line into the corresponding machine code, it is much slower than compiler and assembler. Hence execution time is longer compared to the other language processors. But it is easier for the interpreter to stop in the middle of the execution and do the modifications and debugging if required. Interpreters are commonly installed in web servers as it has a set of executable scripts which needs to be interpreted one by one. It can also be used during the development stage of a program where small chunks of code need to be tested rather than the whole program at once.

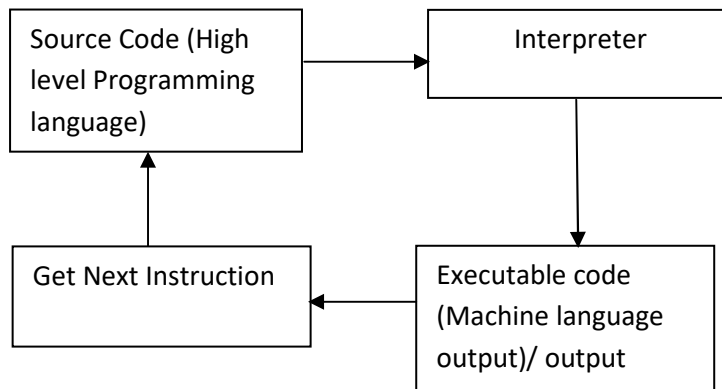


Fig 2: Interpreter

- **Compiler:** Compiler translates a source code written in high level language program into machine codes as a whole in one go. The source code is executed successfully only if it is free from errors. The errors have to be corrected for successful completion. Errors are specified with the line numbers in the source code so that the user can easily go through it and rectify the errors. High level languages like C, C++ and C# use compiler.

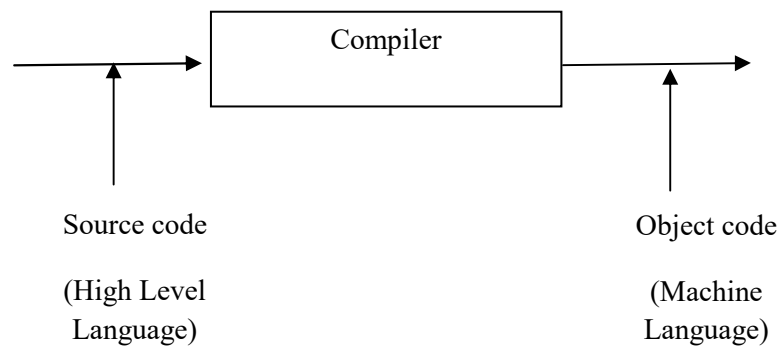


Fig 3: Compiler

Differences between Compiler and Interpreter

1. Compiler reads the whole program all at once while interpreter reads the whole program line by line.
2. Compiler directly generates machine codes easily understood by the computer while interpreter generates only intermediate code after translation.
3. Compiler is time consuming as it takes a lot of time in analyzing the source code while interpreter takes very less time in analyzing the source code.
4. Compiler is faster than interpreter
5. Compiler requires more memory space whereas interpreter requires less memory space.
6. Compiler is used by programming language like C,C++ while interpreter is used by programming languages like Python, Ruby.

3.4 SYNTAX, SEMANTICS AND VIRTUAL COMPUTERS

Syntax and Semantics: The set of rules that defines the structure of the program and format of a language is called the syntax of a program. Syntax tells the computer how to read and write the code with a specific set of words and phrases in a specific order so that the computer can understand when instructions are given by the user. Keywords, operators, punctuation and formatting conventions are encompassed in the syntax of a programming language. Syntax forms the basic foundation of all programming languages and acts

as a bridge between human-readable code and machine-executable instructions. On the other hand, semantics is the meaning associated with each statement of the program language. Semantics is used as a tool for language design for expressing design choices, understanding language features and how they interact. Thus it describes the properties of a language. Every programming language will have its own syntax as well as semantics.

Let's see the syntax and semantics of the Java language.

The structure of a Java program is depicted in Fig 4:

Documentation Section (Suggested)
Package Statement (Optional)
Import Statements (Optional)
Interface Statements (Optional)
Class Definitions (Optional)
Main Method Class { Main Method Definition } (Essential)

Fig 4: General Structure of a Java program

Let's go through each and every section of the Java program

Documentation Section: The name of the program, the author and other details of a program are written as comments in the documentation section which the programmer might refer at a later stage. Comments explain in details about the classes and the algorithm used in the program. Single comments in Java starts with a // and are given in the following form

//This is a comment

Multiline comments are written in this form

```
/* These are comments*/
```

Anything written in between `/*` and `*/` is ignored by Java.

Package Statement: Package statement is the first statement of a Java program. This statement declares the name of the package and informs the compiler it tells the compiler that the classes defined here belong to this package. It is written in the following form

```
package sample;
```

Here sample is the name of a package. This statement is optional. As class defined in the program may not be part of any package.

Import Statement: After package there may be any number of import statements before the class definition. Import statement is of the form:

```
import sample.student;
```

This statement instructs the interpreter to load the student class from the package sample.

Interface Statements: Interface statements are used when we want to implement multiple inheritance feature in the program. This statement is also optional. It consists of a group of method declarations.

Class Definitions: There may be more than one class definitions in a Java program. Class is the main element of a Java program.

Main Method: Main method is the main starting point of a Java program. It creates objects of various classes and establishes communications between them. When the end of main method is reached, the program terminates and the operating system takes back the control of the program.

An example of a simple Java program is illustrated below:

```
class Student
{
    public static void main(String args[])
    {
```



```
        System.out.println ("Java is better than C++");  
    }  
}
```

Now let's explain each statement in details:

Class Student declares a class. All class definitions in Java is written within the curly braces. The line

```
public static void main (String args[])
```

defines a method named main. Here

Public: It is an access specifier that declares the main method as unprotected and therefore it is accessible to all other classes.

Static: The static keyword declares this method a one that belongs to the entire class and it is not a part of any object of a class.

Void: Void states that the main method does not return any value.

As a whole this statement declares a parameter named args which is an array of class string.

The last statement of the program

```
System.out.println ("Java is better than C++");
```

is similar to printf statement in C and cout in C++. It simply prints the line. "Java is better than C++".

Every Java statement is terminated with a semicolon.

Check Your Progress I

1. Assembly language use certain codes called _____ to carry out the different tasks
2. Programming languages like Python, Ruby and Javascript uses _____
3. _____ forms the basic foundation of all programming languages
4. Every programming language will have its own syntax as well as _____
5. _____ statement in Java instructs the interpreter to load the specified class from the package sample.
6. Every Java statement is terminated by a

Virtual Machines: The task of a compiler is to translate the source code into machine code. Java compiler also does the same thing. But Java maintains architecture neutrality. This is possible because the compiler of Java produces an intermediate code known as bytecode for a machine that doesn't really exist. This machine is called the Java Virtual Machine and it resides only inside the computer memory. It is also a computer but a simulated one. It performs all the major functions of a real computer. The process of compilation of a Java program is shown in Fig 5:

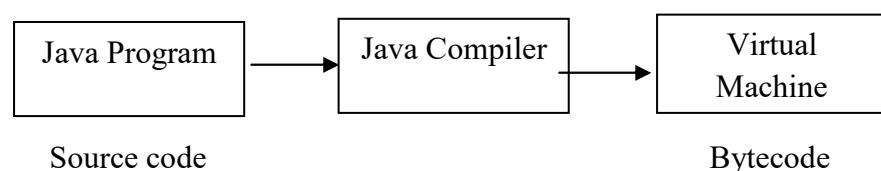


Fig 5: Process of Compilation

The bytecode produced by the virtual machine is not machine specific. So, the machine specific code is generated by the Java interpreter which plays the role of an intermediary between the

virtual machine and the real machines. Every machine has its own interpreter. No two machines can have the same interpreter. Fig 6 depicts how the bytecode generated by the virtual machine is converted into the machine code by the Java interpreter.

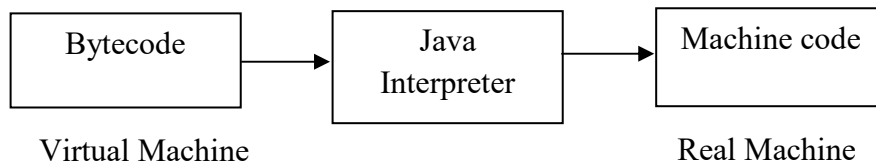
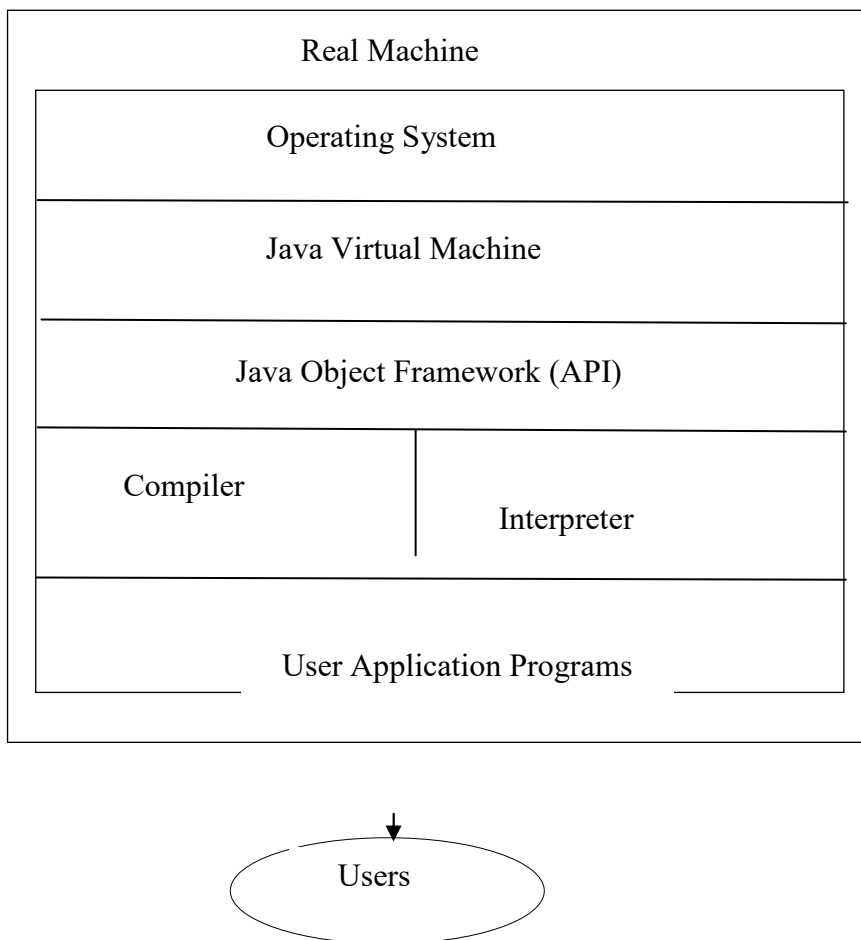


Fig 6: Process of converting bytecode into machine code

The Java API acts as the intermediary between the user programs and the virtual machine which in turn again acts as the intermediary between the operating system and the Java object framework. Fig 7 illustrates how Java works on a typical computer.



3.5 BINDING AND BINDING TIME

Binding in Java refers to the process of associating a method call or variable reference with its corresponding implementation or data type. Binding can occur in run time as well as also in compile time. Run time binding is also called late binding or dynamic binding whereas compile time binding is also called early or static binding. For writing efficient, flexible and maintainable Java program, it's very necessary for the developer to know the differences between the above said two types of binding. The time at which the binding takes place in a program is called binding time. Now let's discuss the two types of binding:

- 1. Static Binding:** Static binding (also known as early binding or compile time binding) takes place during the compile time of a program. It refers to the association of method calls and variable references with their corresponding implementations and data types based on the information available in the source code. The main characteristic of static binding is that the binding is determined and also fixed at compile time. The example below illustrates static binding.

```
public class Main {  
    void addNum (int n1, int n2){  
        System.out.println(n1 + n2);  
    }  
  
    void addNum (intn1, intn2, int n3){  
        System.out.println(n1 + n2 + n3);  
    }  
  
    public static void main (String args[]){  
        //creating Main class object  
        Main object = new Main();  
        //method call  
        object.addNum(20, 30);  
        object.addNumbers(50, 80, 100);  
    }  
}
```

Here the output will be 50 and 230.

Advantages:

1. Static binding is fast compared to dynamic binding as the compiler has all the information it needs to bind the method calls and references. Thus, this is how it contributes towards the overall efficiency of the program.
2. It can detect the errors at an early stage. The inconsistencies or mismatches found in method calls or variables are detected and marked as compile time errors.

Disadvantage

As the name itself indicates that in static binding the decisions made during the binding time remains constant throughout the program. This might be a problem when certain programs require dynamic behaviour. So, static binding lacks flexibility.

2. Dynamic Binding

Dynamic binding (also known as late binding or runtime binding) occurs during the execution phase of the program. Dynamic binding determines the method calls and variable references based on the actual type of the objects at runtime. Due to this dynamic behavior, Java is flexible and polymorphism as well as inheritance can be implemented thus making it a very powerful programming language. The example below illustrates dynamic binding:

```
class Test1 {  
    void show(){  
        System.out.println("Inside show method of Test1 class");  
    }  
}  
  
public class Main extends Test1 {  
    void show () {  
        System.out.println("Inside show method of Main class");  
    }  
  
    public static void main (String args[]){  
        //creating object  
        Test1 object = new Main();  
        //method calls  
        object.show();  
    }  
}
```

The following line will be printed as an output of this program:

Inside show method of Main class

Advantages:

1. Dynamic binding allows polymorphism thus enhancing code reusability and maintainability.
2. In dynamic binding, decisions are made during the run time. The decisions are not fixed at the very beginning and hence allows adaptability in the program.

Disadvantages:

1. Dynamic binding results in slower execution compared to static binding as the runtime has to look up the method dynamically. As a result performance overhead increases.
2. In case of Java dynamic binding can lead to higher memory consumption.

3.6 SUMMING UP

- A language processor is a computer program that translates source code from one programming language into other language
- Assembler is used to translate a program written in assembly language into machine language
- Interpreter translates a single line in the source program into machine code and executes it immediately before proceeding towards the next line.
- Compiler translates a source code written in high level language program into machine codes as a whole in one go.
- The set of rules that defines the structure of the program and format of a language is called the syntax of a program.
- Java compiler translates the source code into an intermediate code called the bytecode for a machine that doesn't really exist.
- Java virtual machine is also a computer but a simulated one and can perform all the tasks of a computer.

- The intermediate code generated by the Java compiler is converted into machine code by the Java interpreter.
- Binding in Java refers to the process of associating a method call or variable reference with its corresponding implementation or data type.
- Binding can occur in run time as well as also in compile time.
- Static binding takes place during the compile time of a program.
- Dynamic binding occurs during the execution phase of the program.

3.7 ANSWERS TO CHECK YOUR PROGRESS

1. mnemonics
2. interpreter
3. Syntax
4. semantics
5. import
6. semi colon

3.8 POSSIBLE QUESTIONS

1. What is a language processor?
2. What is an assembler?
3. How assembler is different from compiler?
4. Distinguish between compiler and interpreter.
5. What do you mean by syntax and semantics of a program?
6. Explain the general structure of a Java program.
7. What is a Java Virtual Machine? Explain the process of compilation of a program in Java.
8. What is binding?
9. What is binding time?
10. Explain static and dynamic binding with the help of an example.
11. List two advantages each for static binding and dynamic binding.
12. List the demerits of dynamic binding.

3.9 REFERENCES AND SUGGESTED READINGS

- <https://www.w3schools.blog/static-vs-dynamic-binding-in-java>
- <https://www.cl.cam.ac.uk/teaching/0809/Semantics/notes-mono.pdf>
- <https://medium.com/@appwebcoders/compile-time-binding-and-run-time-binding-in-java-2e1ef107d586>
- Balaguruswamy, E. (2014). *Programming with Java-A Primer*. McGraw-Hill Professionals.

---X---

UNIT 4: INTRODUCTION TO IMPERATIVE PROGRAMMING LANGUAGES

Unit Structure:

- 4.1 Introduction
- 4.2 Unit Objectives
- 4.3 Imperative Programming Language
- 4.4 Statements and data types
- 4.5 Subprograms, sequence control and data control
- 4.6 Dynamic allocation using pointers
- 4.7 Operating and Programming environment
 - 4.7.1 Java Runtime Environment
 - 4.7.2 Application Programming Interface
- 4.8 Summing Up
- 4.9 Answers to check your progress
- 4.10 Possible Questions
- 4.11 References and Suggested Readings

4.1 INTRODUCTION

Imperative programming refers to a paradigm of computer programming in which a program is described with the help of a sequence of steps which results in changing the state of the computer. It is a model based on moving bits around and changing the state of the machine. In this type of programming, natural language can also be used in expressing the commands to be given to the computer. Imperative programming focusses on how to accomplish a task rather than what to accomplish as in declarative programming.

4.2 UNIT OBJECTIVES

After going through this unit you will be able to:

- Understand Imperative programming language
- Learn statements and data types in imperative programming language
- Learn subprograms, sequence control and data control

- Understand the concept of dynamic memory allocation using pointers
- Understand the programming environment of the imperative programming language.

4.3 IMPERATIVE PROGRAMMING LANGUAGES

Imperative or procedural languages are command-driven or statement-oriented language. The basic concept is the state of the machine which is nothing but the set of all memory locations in the computer. As we all know that a program is a sequence of statements and after execution of each statement the computer changes value of one or more locations in its memory to enter to a new state. This programming model is supported by C, C++, Java, FORTRAN, ALGOL, PL/I, Pascal, Ada, Smalltalk and COBOL. The different imperative languages can further be assigned to three subordinate programming style: structured, procedural and modular.

Structured programming consists of specific control structures such as sequences, selection and iteration. It is easy to maintain, read and understand a structured program. It promotes code reuse, since even internal modules can be extracted and made independent, residents in libraries, described in directories and referenced by many other applications. Almost all high-level language program support structured programming. Procedural programming divides the task a program is supposed to perform into smaller sub-tasks, which are individually described in the code. This results in programming modules which can also be used in other programs. Modular programming model designs, develops and tests the individual program components independently of one another. The individual modules are then combined to create the actual software.

In this Unit we will use the language Java for understanding the concepts of the Imperative programming language.

4.4 STATEMENTS AND DATA TYPES IN IMPERATIVE PROGRAMMING LANGUAGE

The statements in Imperative programming language can be mainly divided into three types:

- **Selection Statements:** In selection statements there are two or more execution path, out of which one is selected and executed accordingly. It is mainly used for decisions and branching such as choosing between multiple paths. In Java, there are four selection statements: if, if-else, if-else-if-else and switch statement. The syntax followed by an example is given below.

a) **if statement:** if(condition)

```
{
// Statements to execute if
// condition is true
}
```

Example: class If Statement {

```
public static void main(String[] args) {
```

```
int number = 10;
```

```
// checks if number is less than 0
```

```
if (number < 0) {
```

```
System.out.println("The number is negative.");
```

```
}
```

```
System.out.println("Statement outside if block");
```

```
}
```

```
}
```

b) **if-else statement:** if(condition)

```
{
```

```
// code if condition is true.
```

```
}
```

```
else
```

```
{
```

```

        //code if condition is false.
    }

```

Example: class Main {

```

    public static void main(String[] args) {

```

```

        int number = 10;

```

```

        // checks if number is greater than 0

```

```

        if (number > 0) {

```

```

            System.out.println("The number is positive.");

```

```

        }

```

```

        // execute this block

```

```

        // if number is not greater than 0

```

```

        else {

```

```

            System.out.println("The number is not positive.");

```

```

        }

```

```

        System.out.println("Statement outside if...else block");

```

```

    }

```

```

}

```

c) if-else-if-else: if (condition1)

```

{

```

```

    //block of code to be executed if condition1 is

```

```

true

```

```

} else if(condition2)

```

```

{

```

```

        // block of code to be executed if condition1 is false
and condition2 is true

    }

    else

    {

        //block of code if condition1 is false and condition2
is false

    }

```

Example: class Main {

```

    public static void main(String[] args) {

        int number = 0;

        // checks if number is greater than 0
        if (number > 0) {

            System.out.println("The number is positive.");

        }

        // checks if number is less than 0
        else if (number < 0) {

            System.out.println("The number is negative.");

        }

        // if both condition is false
        else {

            System.out.println("The number is 0.");

        }

    }

}

```

```

d) switch statement: switch(expression){
                                case x: // block of code
                                    break;
                                case y: // block of code
                                    break;
                                default:// block of code
                                    }

```

Example:

```

class Main {
    public static void main(String[] args) {

        int number = 44;
        String size;
        // switch statement to check size
        switch (number) {
            case 29: size = "Small";
                    break;
            case 42: size = "Medium";
                    break;
            case 44: size = "Large";
                    break;
            case 48: size = "Extra Large";
                    break;
            default: size = "Unknown";
                    break;
        }
    }
}

```

```

    }

    System.out.println("Size: " + size);

    }

}

```

- **Sequence Statements :** Sequence statement consists of a block of statements that are executed one after the other sequentially

Example:

```
class HelloWorld
```

```

    {

        public static void main(String args[])

        {

            System.out.println("Hello World"); //Prints on screen "Hello
World"

        }

    }

```

These are simply a block of statements that are being executed one after the other.

- **Iteration Statements:** Iteration statements repeat a certain block of statements in a loop until it keeps on satisfying a certain condition. The three iteration statements are: for loop, while loop and do-while loop.

a) **for loop:** for (initialization expr; test expr; update exp)

```

{
    // body of the loop
    // statements we want to execute
}

```

Example:

```
class sample{
```

```

        public static void main(String[] args)
        {
            for (int i = 1; i<= 10; i++) {
                System.out.println(i);
            }
        }
    }
}
b) while loop: while(testExpression){
    //body of loop
}

```

Example:

```

class Main {
    public static void main(String[] args) {

        // declare variables
        int i = 1, n = 5;

        // while loop from 1 to 5
        while(i<= n) {
            System.out.println(i);
            i++;
        }
    }
}

```

```

c) do-while loop:do{
    //body of the loop
}while(testExpression)

```

Example:

```

import java.util.Scanner;
class Main {
    public static void main(String[] args) {

        int i = 1, n = 5;

        // do...while loop from 1 to 5
        do {
            System.out.println(i);
            i++;
        } while(i<= n);
    }
}

```


The different data types in Imperative programming language can be mainly divided into types:

- **Primitive Data Types:** The most fundamental data types usable in a programming language are the primitive data types. There are eight primitive data types: Boolean, byte, character, short, int, long, float, *and* double. These data types in any programming language serves as the foundation for data manipulation
- **Derived Data Types:** Derived data types are derived from the primitive data types. These includes String, array and classes.
- **User-Defined Data Types:** These are the data types that are defined by the user depending upon the requirement of the program.

CHECK YOUR PROGRESS

1. Give two examples of Imperative programming languages.
2. Structured programming consists of specific control structures such as _____, selection and iteration
3. What is modular programming?
4. Selection statements are used for decisions and branching such as choosing between _____ paths.
5. The three iteration statements are for loop, _____ loop and do-while loop.
6. Array is a _____ data type.
7. _____ data type serves as the foundation for data manipulation

4.5 SUBPROGRAMS, SEQUENCE CONTROL AND DATA CONTROL

Subprogram: A subprogram is a set of instructions or a piece of program that can be called from different parts or places of a program. It eliminates the process of writing the same code again and again wherever it needs to be executed. The programmer can call this subprogram anywhere in the program without bothering

about how to implement the operation. Subprograms are also called subroutines. There are mainly three types of subprograms: procedure, functions and methods.

The general characteristics of a subprogram are:

1. Subprogram always have a single entry.
2. At any moment only one subprogram can be executed, others are suspended but they may remain active.
3. The execution returns to the main program or the caller once the subprogram terminates.

Subroutines in JAVA are defined in a class. A subroutine can only be called if it is defined somewhere in a program. The name of the subroutine, information required to make call to the subroutine, and the code that will be executed each time the subroutine is called must be defined in the subroutine. A subroutine definition in Java takes the form:

```
modifiers return_type subroutine_name(parameter list){  
  
statements  
  
}
```

The statements within the curly braces is the body of the subroutine. Modifiers define some characteristics of the subroutine such as whether it is static or public. return_type refers to a method that may return a value or void. Parameters represent information that is passed into the subroutine from outside such as data type, order or number of parameters to be used by the subroutine's internal computations. Sometimes there may be zero parameters also.

Example:

```
public class Factnum{  
  
public int Fact(int n) //Subroutine method definition  
  
    {  
  
        int f=1;  
  
        for ( int=1;i<=n; i++)
```

```

        f= f*I;

        return f;

    }

public static void main(String args[])

{

    Scanner sin = new Scanner(System.in);

    int inp = sin.nextInt();

    int factorial= Fact(n1); //Subroutine call

    System.out.println(factorial);

}}

```

Sequence control: When a subprogram or subroutine is called, the execution of calling program is temporarily stopped during execution of the subprogram. Once the subprogram is completed, execution of the calling program resumes at the point immediately following the call.

The following assumptions are made for simple subroutine call and return structure

- i) Subprogram can never be recursive
- ii) Explicit call statements are required
- iii) Subprograms must execute completely at call
- iv) Immediate transfer of control at point of call or return
- v) Single execution sequence for each subprogram

4.6 DYNAMIC ALLOCATION USING POINTERS

Heap memory also known as dynamic memory in JAVA is allocated automatically when a function is called and deallocated automatically when a function exits. Whenever a programmer requests for memory it is allocated a block of memory of a particular size and it remains until something happens that makes it go away. The significant source of errors in C/C++ is dropping all

references to a memory location without deallocating it. This error is known as memory leak. Java removes this error by handling memory deallocation automatically, using garbage collection.

Heap memory is a huge memory available for use by the programs. Whenever a program needs memory, it makes an explicit request by executing the heap *allocation* function. The allocation function reserves a block of memory of the requested size and returns a pointer to the program. For example a program makes three separate allocation requests for storing three GIF images. In this case the allocation function will allocate three blocks of heap memory for the three images and will return three pointers to the main program. The pointers serve as the base addresses of the heap memory. At any time some blocks might be in use and some blocks will be free for use as they are not committed yet by any program. So heap memory is available to satisfy allocation requests. The heap manager keeps track of the memory being allocated to the different programs along with the information about the free blocks. As soon as a new program is allocated a block of heap memory, the heap manager updates its private data structure.

As soon as a program finishes its task, the block of memory used by the program is marked unused. This allows the garbage collector of JAVA to clean the memory space marked as unused and make it available for future allocation request. The heap manager again updates its private data structure to show that that area of memory is free again and can be allocated to new program request.

Let's go through a simple heap example. Here is a simple example that allocates an Employee object block in the heap, and then deallocates it. This is the simplest possible example of heap block allocation, use, and deallocation. The example shows the state of memory at three different times during the execution of the above code. The stack and heap are shown separately in the drawing—a drawing for code which uses stack and heap memory needs to distinguish between the two areas to be accurate since the rules which govern the two areas are so different. In this case, the lifetime of the local variable empPtr is totally separate from the lifetime of the heap block, and the drawing needs to reflect that difference.

```
void Heap1() {  
    Employee empPtr;  
    // Allocates local pointer local variable (but not its pointee)
```

```

// T1
/ Allocates heap block and stores its pointer in local variable.
// Dereferences the pointer to set the the name to Sam
empPtr = new Employee();
empPtr.setName("Sam");
// T2
// Deallocate heap block makes the pointer equals to null.
//This will let the garbage collection to know that this object is
used and it
//must be cleared from the head
// The programmer must remember not to use the pointer
// after the pointee has been deallocated (this is
// why the pointer is shown in gray).
empPtr = null;
// T3
}

```

4.7 OPERATING AND PROGRAMMING ENVIRONMENT

Java is an object oriented programming language that enables us not only to organize program codes into logical units called objects but also to take advantage of encapsulation, inheritance and polymorphism. Java programming environment consists of a programming language, an API specification and a virtual machine specification.

4.7.1 Java Runtime Environment

The Java Runtime Environment, or JRE, is the software environment that runs on top of a computer's operating system software and provides the class libraries and other resources that a specific Java program requires to run. The three interrelated components that are required for developing and running Java programs are JRE, JDK and JVM. Let's study briefly about JDK and JVM.

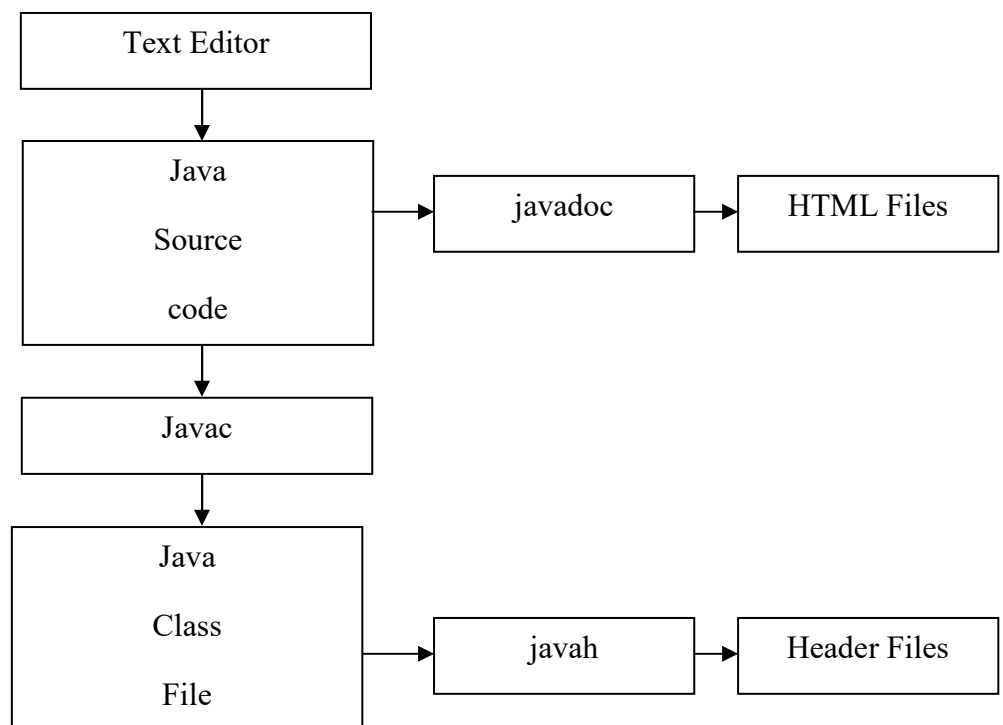
- The Java Development Kit, or JDK, is a large number of development tools required for developing Java applications. The tools and their descriptions are given in table 1:

Tool	Description
appletviewer	Enables us to run Java applet
java	Java interpreter which runs applets and applications by reading and interpreting bytecode files
javac	The Java compiler which translates Java source code to bytecode files that the interpreter can understand
javadoc	Creates HTML-format documentation from Java source code files
javah	Produces header files for use with native methods
javap	Java disassemble which enables us to convert bytecode files into a program description
jdb	Java debugger, which helps us to find errors in our programs

Table 1: Java Development Tools

- The Java Virtual Machine, or JVM, runs live Java applications and is the one that calls the main method present in a Java code. JVM is a part of JRE(Java Runtime Environment).

Figure 1 depicts the process of building and running java application program.



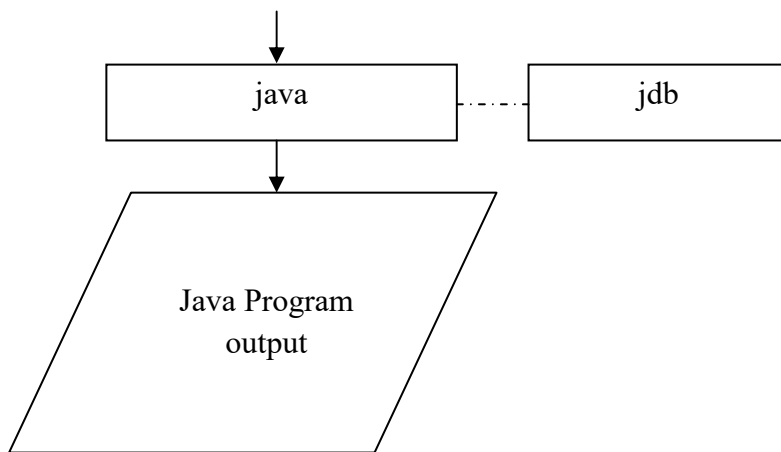


Fig 1: Process of building and running Java application programs

4.7.2 Application Programming Interface

Java Application Programming Interface (API) or Java Standard Library (JSL) includes hundreds of classes and methods grouped into several functional packages. The most commonly used packages are:

- **Language Support Package:** A collection of classes and methods required for implementing basic features of Java
- **Utilities Package:** A collection of classes to provide utility functions such as date and time functions.
- **Input/Output package:** A collection of classes required for input/output manipulation.
- **Networking package:** A collection of classes for communicating with other computers via Internet.
- **AWT:** The Abstract Window Tool Kit package contains classes that implements platform independent graphical user interface.
- **Applet package:** This includes a set of classes that allows us to create Java applets.

4.8 SUMMING UP

- Imperative or procedural languages are command-driven or statement-oriented language where the basic concept is set of all memory locations in the computer.

- A program is a sequence of statements and after execution of each statement the computer changes value of one or more locations in it's memory to enter to a new state.
- C, C++, Java, FORTRAN, ALGOL, PL/I, Pascal, Ada, Smalltalk and COBOL are all imperative programming languages.
- Structured programming consists of specific control structures such as sequences, selection and iteration.
- Procedural programming divides the task a program is supposed to perform into smaller sub-tasks, which are individually described in the code.
- Modular programming model designs, develops and tests the individual program components independently of one another.
- The statements in imperative languages can be categorized into: selection, sequence and iteration statement.
- The different data types in imperative programming language are : primitive, derived and user defined data types.
- A subprogram is a set of instructions or a piece of program that can be called from different parts or places of a program.
- When a subprogram or subroutine is called, the execution of calling program is temporarily stopped during execution of the subprogram.
- In Java allocation and deallocation of memory is done automatically using garbage collection.
- The three interrelated components that are required for developing and running Java programs are JRE, JDK and JVM.
- Java Application Programming Interface (API) or Java Standard Library (JSL) includes hundreds of classes and methods grouped into several functional packages

4.9 ANSWERS TO CHECK YOUR PROGRESS

1. C++ and Java
2. Sequences

3. Modular programming model designs, develops and tests the individual program components independently of one another. The individual modules are then combined to create the actual software.
4. Multiple
5. while
6. Derived
7. Primitive

4.10 POSSIBLE QUESTIONS

1. What is imperative programming language?
2. What are the three main programming style? Explain.
3. Explain the selection statement type with the help of an example.
4. Distinguish between sequence and iteration statement with the help of examples.
5. What are the different data types in imperative programming language? Explain.
6. Define subprogram.
7. Mention the general characteristics of subprogram.
8. Write a short note on: Dynamic allocation, Garbage Collection
9. What is a heap memory? How does it help in overcoming the memory leak program?
10. Describe the programming environment of Java.

4.11 REFERENCES AND SUGGESTED READINGS

- <https://www.slideshare.net/slideshow/java-methods-or-subroutines-or-functions/249830971>
- <https://math.hws.edu/javanotes/c4/s2.html>

- 3.https://ggn.dronacharya.info/csedep/Downloads/QuestionBank/Even/IV%20sem/Section_A_Sequence_Control.pdf
- https://opensa.cs.vt.edu/ODSA/Books/vt/cspointer/fall-2017/Pointers_Test/html/HeapMem.html
- <https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqb/index.html>
- <https://www.ibm.com/think/topics/jre>
- Balaguruswamy, E. (2014). *Programming with Java-A Primer*. McGraw-Hill Professionals.

---X---

UNIT 5: Concept of Subprogram in Imperative Programming Language

Unit Structure:

- 5.1 Introduction
- 5.2 Objectives
- 5.3 Subprogram activation- parameter passing methods
- 5.4 Scope rules for names
- 5.5 Nested procedures and Syntax
- 5.6 Summing Up
- 5.7 Answers to Check Your Progress
- 5.8 Possible Questions
- 5.9 References and Suggested Readings

5.1 INTRODUCTION

In the previous chapter we have learned about the basic concept of subprogram, characteristics of a subprogram, subprograms in Java and also the sequence control in subprogram. In this chapter we will learn about the different parameter passing methods in subprograms. We will also learn about scope rules for names as well as about the nested procedures, syntax and translation.

5.2 OBJECTIVES

After going through this unit, you will be able to

- Understand how the parameters are passed in a subprogram
- The two ways of parameter passing: pass by value and pass by reference
- Different scope rules of variables
- Different nested procedures and syntax

5.3 SUBPROGRAM ACTIVATION- PARAMETER PASSING METHODS

The process of calling and executing a subprogram (such as a function or procedure) in a program is referred to as subprogram activation. For smooth execution of a program, certain actions are required during the activation. Such type of actions includes allocating resources, passing parameters, and ensuring that control returns to the correct location after the subprogram completes.

The key steps involved in a subprogram activation are:

1. **Subprogram call:** A subprogram is invoked or called by the main program or another subprogram by using a function call, procedure call, etc.
2. **Creation of stack:** A new stack frame is created once a subprogram is activated. This frame contains the local variables, parameters to be passed to the subprogram and the return address as where to start the execution once a subprogram terminates.
3. **Parameter Passing:** Parameters passing is the arguments that are expected by the subprogram and are passed by means of value, reference and name.
4. **Execution:** The parameters that are passed along with the local variables are used during the execution of a subprogram.
5. **Return:** The stack frame is destroyed once the subprogram completes its execution and the control returns to the main program.
6. **Control Transfer:** Once the control is transferred back to the to the point where the subprogram was called, the main program starts resuming again.

Different programming paradigms (procedural, object-oriented) and languages (C, Java, Python) handle these aspects differently, but the concept of subprogram activation remains consistent. We will discuss how the parameters are being passed during subprogram activation.

As we have already learned about the basic concepts of a subprogram, let's try to understand the parameter passing methods of a subprogram. Before discussing about the parameter passing methods we need to understand how a function is defined. A program consists of a number of procedures and functions. These functions which are called subprograms may or may not contain parameters. A function is defined in the following way in Java:

```
public static double average(double a, double b, double c){
// code
}
```

Here double is the data type and average is the function name. x, y, z are the parameters.

Now let's write a subprogram for this function:

```
public static double average(double a, double b, double c)
{
return(a + b + c)/3;
}
```

So the program will be like:

```
import java.util.Scanner;
public class Exercise2{

public static void main(String[]args)
{
Scanner in =new Scanner(System.in);
System.out.print("Input the first number: ");
double x =in.nextDouble();
System.out.print("Input the second number: ");
double y =in.nextDouble();
System.out.print("Input the third number: ");
double z =in.nextDouble();
System.out.print("The average value is "+average(a,b,c)+"\n");
}

public static double average(double a, double b, double c)
{
return(a+b+c)/3;
}
}
```

A subprogram may be defined before or after the main function. The parameters in a subprogram can be passed basically in two ways:

- Pass by value
- Pass by reference
- Pass by value: Pass by value is the default mode of passing parameters into methods. This means that the parameters

declared inside the method body is a copy of the original argument that was passed in. Whenever a method is called, a copy of each actual parameter is passed. Now if we make any change to the copy, there will be no effect on the actual parameters. In Java there is no mechanism to change the actual parameters value. A Java program where the parameters are passed by value is demonstrated in the program below:

Program to demonstrate pass by value

```
public class Sample1 {  
  
    public static void main(String[] args) {  
        int number=10;  
        System.out.println("Before: " + number);  
        modifyNumber(number);  
        System.out.println("After: " + number);  
    }  
    public static void modifyNumber(int value) {  
        value = value * 2;  
        System.out.println("Modified: " + value);  
    }  
}
```

- Pass by reference: In pass by reference method the actual value is not passed rather an alias or the reference to the parameter is passed. Here if we make any changes to the parameter instances, it would affect the actual value also. Java is always a pass by value mechanism but there are ways to make it pass by reference. The pass by reference is handled by pass by value in Java.

To understand the above concept let's first understand the storage mechanism of Java. The reference variables along with the names of methods and classes are stored in heap but the primitive data types are stored always in stack along with their values.

As reference variables are stored in stack, so the value of the parameter of such variables is the reference or address of the given variable. This is in case of arrays, objects and strings.

For example, if we have an array 'a' with elements {1, 2, 3} and we pass 'a' as a parameter to a method, then the method receives the

copy of reference or address of 'a' as its parameter. The following example demonstrates the use of pass by reference in Java.

Example:

```
import java.util.*;
public class ParamPass{
    public static void changeArray(int[]a){
        A[0]=a[0]*2;
        System.out.println("Inside
method:a="+Arrays.toString(a));
    }
    public static void main(String[] args){
        int[] array = {1,2,3};
        System.out.println("Before calling changeArraymethod:arr = "+
Arrays.toString(array));
        changeArray(array);
        System.out.println("After calling changeArraymethod:arr = "+
Arrays.toString(array));
    }
}
```

After executing this program we will get :

Before calling changeArray method: arr = [1, 2, 3]

Inside method: a = [2, 2, 3]

After calling changeArray method: arr = [2, 2, 3]

5.4 SCOPE RULES FOR VARIABLES

The part of the program where the variable is accessible is called the scope of a variable. Scope defines the visibility and the lifetime of a variable. Java allows variable to be declared not only in the main () method but also within any block. A block begins with a curly brace and ends with a curly brace. Thus, a block defines a scope. A scope determines what objects are visible to other parts of your program. In other programming languages the two general categories of scope are: local and global. But these two scopes doesn't fit the strict object oriented Java language.

In Java, here are four scopes for variables in Java: local, instance, class, and method parameters.

1. Local Variables: The variables that are declared inside a method, constructor, or code block are referred to as local variables. These variables can only be accessed within the block in which they are defined. The lifetime of local variable exists only while the method or block is executing.

Example: public class Main{

 public static void main(String[] args) {

 int x=10; // 'x' is a local variable

 System.out.println(x);

 }

}

2. Instance Variables: The variables that are inside a class but are outside any method are called instance variables. These variables are initialized at the time of class instantiation. Instance variables can be accessed by methods and constructors only of that particular class. The lifetime of this class variable exists as long as the object that contains it is alive.

Example: class Exercise{

 int speed;

 public void setSample(int sample){

 this.sample = sample;

 }

}

3. Class Variables: The variables that are declared inside the class are called class variables. They are declared with the static keyword and are outside any method. Class variables are accessible within static methods or from any object of the class. Static variables can be accessed using the class name or object reference. The lifetime of class variable exists as long as the class is loaded in the JVM.

Example: class Maths{

 static x=0;


```

public static void increment(){

    x++;

}

}

```

The scope defined by a method begins with curly braces and if the method consists of parameters, they are too included within the block's scope. A variable that is declared inside a scope cannot be visible to the code that is defined outside the scope. Variables declared inside the scope are localized and also protected from unauthorized access and modification. Scopes can also be nested. A new nested scope is created as soon as a new block is created. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true.

Let's understand the effect of nested scopes, with the help of an example.

// Demonstrate block scope.

```

class Scope {

    public static void main(String args[]) {

        int a; // known to all code within main

        a = 10;

        if(a == 10) { // start new scope

            int b = 20; // known only to this block

            // a and b both known here.

            System.out.println("a and b: " + a + " " + b); a = b * 2;

        }

        // b = 100; // Error! b not known here

        // a is still known here.

        System.out.println("a is " + a);

    }

}

```

```
}
```

As we can observe, the variable `a` is declared at the start of `main()`'s scope and is accessible to all subsequent code within `main()`. The variable `b` is declared within the `if` block and so `b` is only visible to other code within its block. Therefore, `b` is not known outside the block. So, if `b` was declared outside the block, a compile time error would have occurred. Within the `if` block, `a` can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

A variable is valid only after its declaration. So, a variable is usually defined at the start of a method so that it can be used by the other codes within the block and it's useless to declare the variable at the end of the block as no code will be able to access it.

A variable is created once it enters a scope and is destroyed as it leaves the scope. So, this indicates that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

In a variable declaration if an initializer is included, then that variable will be reinitialized each time the block in which it is declared is entered. Let's understand this with the help of an example.

For example, consider the next program.

```
// Demonstrate lifetime of a variable.

class LifeTime{ public static void main(String args[])
{
    int x;

    for(x = 0; x < 3; x++) {

        int y = -1; // y is initialized each time block is entered
        System.out.println("y is: " + y); // this always prints -1

        y = 100;
    }
}
```

```
        System.out.println("y is now: " + y); }  
    }  
}
```

The output generated by this program is shown here:

```
y is: -1  
y is now: 100  
y is: -1  
y is now: 100  
y is: -1  
y is now: 100
```

As we can see, `y` is reinitialized to `-1` each time the inner for loop is entered. Even though it is subsequently assigned the value `100`, this value is lost. Although blocks can be nested, a variable cannot be declared with the same name as one in an outer scope.

For example, the following program cannot be compiled as it is illegal:

```
// This program will not compile  
  
class ScopeErr { public static void main(String args[]) {  
  
    int num = 1; {  
  
    int num = 2; // Compile-time error as num is already defined!  
  
    }  
  
    }  
  
}
```

We need to understand carefully about the variable scope as it is very crucial for managing data visibility, memory usage, and preventing conflicts in larger Java applications.

CHECK YOUR PROGRESS

1. A program consists of a number of procedures and _____
2. A _____ can be defined before or after the main function.
Subprogram
3. _____ defines the visibility and the lifetime of a variable.
Scope
4. A variable that is declared _____ a scope cannot be visible to the code that is defined outside the scope. Inside
5. A variable declared within a block will lose its value when that _____ is left.

5.5 NESTED PROCEDURE AND SYNTAX

A method of a class can be called only by an object of that class using the dot operator. A method can also be called by another method of the same class using the class name, but only if both methods are present in the same class. This is known as nesting of methods. Efficient code organization and simplified method calls within a class are possible through nesting of methods. A program to illustrate the nesting method in Java is given below:

```
import java.util.Scanner;
public class Nesting_Methods
{
    int perimeter(int l, int b)
    {
        int pr=12*(l + b);
        return pr;
    }
    int area(int l, int b)
    {
        int pr= perimeter(l, b);
        System.out.println("Perimeter:"+pr);
        int ar=6* l * b;
```

```

        return ar;
    }
    int volume(int l, int b, int h)
    {
        int ar= area(l, b);
        System.out.println("Area:"+ar);
        int vol;
        vol= l * b * h;
        return vol;
    }
    public static void main(String[]args)
    {
        Scanner s =new Scanner(System.in);
        System.out.print("Enter length of cuboid:");
        int l =s.nextInt();
        System.out.print("Enter breadth of cuboid:");
        int b =s.nextInt();
        System.out.print("Enter height of cuboid:");
        int h =s.nextInt();
        Nesting_Methods obj=new Nesting_Methods();
        int vol=obj.volume(l, b, h);
        System.out.println("Volume:"+vol);
    }
}

```

Another program demonstrating the use of nested methods in Java is given below:

```
class Nesting
```

```
{
```

```
    int m, n;
```

```
    Nesting(int x, int y)
```

```
    {
```

```
        m = x;
```

```
        n = y;
```

```
    }
```

```
    int largest()
```

```
{
```

```

        if(m>=n)
            return(m);
        else
            return(n);
    }
    void display()
    {
        int large = largest();
        System.out.println("Largest value = "+ large);
    }
}

void display()
{
    int large = largest();
}

}

}

class NestingTest
{
    public static void main(String args[])
    {
        Nesting nest = new Nesting (50, 40);
        nest.display();
    }
}

```

Java programming language is also capable of defining to define a class within another class. Such a class is called a nested class and is shown below:

```
class Class1 {  
    ...  
    class Class2 {  
        ...  
    }  
}
```

Here Class1 is the outer class and Class2 is the inner class.

There are two types of nested classes in Java: non-static and static. Non-static nested classes are called inner classes. Nested classes that are declared static are called static nested classes. These inner classes don't have a name and only a single object is created for it. These subclasses can be helpful while overloading methods of a class or interface without having to subclass a class.

Let's understand the inner class with the help of an example:

// Java program implements method inside method

```
public class nestedpr {  
  
    // create a local interface with one abstract  
    // method sample()  
  
    interface loc_Interface {  
        void sample();  
    }  
  
    // function have implements another function run()  
  
    static void Sam()  
    {  
        // implement run method inside Foo() function  
  
        loc_Interface r = new loc_Interface() {  
            public void sample()  
            {  
                System.out.println("GUCDOE");  
            };  
        };  
        r.sample();  
    }  
}
```

```

public static void main(String[] args)
{
    Sam();
}
}

```

Nested classes are usually used when we need to logically group classes that are used in one place. It also helps to increase encapsulation. Nested classes make a program more readable and maintainable in code.

5.6 SUMMING UP

- A program consists of a number of procedures and functions.
- A subprogram may be defined before or after the main function.
- The parameters in a subprogram can be passed basically in two ways:
 - Pass by value
 - Pass by reference
- Scope defines the visibility and the lifetime of a variable.
- A variable that is declared inside a scope cannot be visible to the code that is defined outside the scope.
- Scopes can also be nested.
- A variable is valid only after it's declaration.
- A variable is created once it enters a scope and is destroyed as it leaves the scope.
- A method can also be called by another method of the same class using the class name, but only if both methods are present in the same class
- Efficient code organization and simplified method calls within a class are possible through nesting of methods.

5.7 ANSWERS TO CHECK YOUR PROGRESS

1. Functions
2. Subprogram

3. Scope
4. Inside
5. Block

5.8 POSSIBLE QUESTIONS

1. What are the key steps in execution of a subprogram? Explain
2. What are the four scopes for variables in Java? Explain
3. How a function is defined in Java?
4. Mention the two ways in which the parameters are passed in a subprogram?
5. Explain pass by value method with the help of an example.
6. Explain pass by reference with the help of an example.
7. Define scope of a variable.
8. What are the two scopes of variable in Java?
9. Explain nesting of methods with the help of an example.
10. What is the benefit of using nested classes?

5.9 REFERENCES AND SUGGESTED READINGS

- <https://www.sanfoundry.com/java-program-shows-nesting-methods/>
- <https://www.tutorialspoint.com/different-ways-to-achieve-pass-by-reference-in-java>
- Schildt, H. (2007). Java: the complete reference.

---x---

BLOCK- II

OBJECT ORIENTED LANGUAGES

Unit 1: Data Abstraction

Unit 2: Inheritance

Unit 3: Polymorphism

Unit 4: Exception Handling

UNIT 1: DATA ABSTRACTION

UNIT STRUCTURE:

- 1.1 Introduction
- 1.2 Objectives
- 1.3 Introduction to Data Abstraction and Encapsulation
- 1.4 Class
- 1.5 Object
- 1.6 Constructor
- 1.7 Destructor
- 1.8 Templates
- 1.9 Summing Up
- 1.10 Possible Questions
- 1.11 References and Suggested Readings

1.1 INTRODUCTION

In 1960, the scientists of Norwegian computing center had first introduced the major concepts of Object-Oriented Programming (OOP) that are class, object and inheritance. They had developed Simula programming language by introducing these OOP concepts in it. In 1970, Xerox Corporation had introduced the concept of OOP by developing the first OOP language, 'Smalltalk'.

Object-Oriented Programming(OOP) is a programming paradigm that stretches more importance on data than the algorithm. Concept of object is utilized to develop OOP where object is a physical entity that combines data and procedures together in a group. Major advantages of OOP are presented as follows.

- Code reusability is improved in OOP. Existing objects can be utilized to derive new classes in OOP.
- Maintenance is simpler and easier in OOP as objects are independent entities. Also due to Inheritance, maintenance can be easily performed in OOP.
- Improved modularity is realized in OOP. In OOP, a large and complex problem can be solved by developing small and manageable objects.

- Data security and integrity can improved in OOP by implementing Encapsulation to provide a mechanism for hiding internal properties of an object from the other objects.
- In OOP, needless implementation details are hidden from the users and as a result development of applications will be simpler.
- In OOP, new features can be easily added without affecting other parts of software. So, it is easier to scale applications in OOP.
- Debugging and testing is easier in OOP. Objects are independent entities and as a result, it will be easier to debug individual objects. Similarly, code testing is also easier as individual objects can be easily tested.

Main properties of Object Oriented Programming (OOP) are Data Abstraction, Encapsulation, Inheritance and Polymorphism. Any programming language that supports these four properties is an OOP language. For example: C++, Java etc. In this unit, we are going to discuss about Data Abstraction and Encapsulation. We will learn about class and objects. Concepts of constructor, destructor and templates are also discussed in this chapter.

1.2 OBJECTIVES

After reading this unit, you are expected to be able to learn:

- About Data Abstraction and Encapsulation in OOP
- Meaning of class and object
- Construction of class in Java and C++
- Instantiation of objects
- About constructor and its types
- About destructor
- About templates

1.3 INTRODUCTION TO DATA ABSTRACTION AND ENCAPSULATION

Data abstraction is one of the main properties of OOP which allows hiding the implementation particulars of objects and offering only the necessary features to the users. This process provides a mechanism for the developer to develop a complex software system by hiding its internal complexities from the other users of that software. As a result a user can utilize the complex software system without realizing its internal complexities.

Data abstraction in OOP can be implemented through Encapsulation. **Encapsulation** is also a main property of OOP which permits combining data and methods together to form a single unit. In such a unit, data are accessed by the available methods to perform their tasks and due to the Encapsulation process, restriction on the data access from outside the corresponding unit can be implemented. So, data security and data integrity can be achieved through Encapsulation in OOP.

1.4 CLASS

In OOP, Encapsulation is implemented by defining classes. A class is a model or a blueprint that forms a single unit by enclosing data and methods. So, class can be used as a language tool to create user-defined data types in OOP.

The syntax to define a class in Java is presented as follows.

```
class ClassName
{
    Body of the class
}
```

The syntax to define a class in C++ is presented as follows.

```
class ClassName
{
    Body of the class
};
```

In the above syntax, 'ClassName' is the name of the class and 'class' is a keyword. The body of a class can include data members with different data types, methods or functions and nested classes. A nested class is a class that is defined inside a class.

In OOP, access specifiers are associated with the members of classes so that visibility and accessibility of the members of a class from outside the class can be controlled. In Java, four access specifiers are available as presented in the following points.

- **public:** Class members declared with the access specifier, 'public' are accessible from any part of the program.
- **private:** Class members declared with the access specifier, 'private' are accessible by only the members of the class where they are declared. It means that these members are not accessible from outside the class where they are declared.
- **protected:** 'protected' access specifier is associated with Inheritance. Class members declared with the access specifier, 'protected' are accessible by the same class members and by the members of the derived classes of the class where they are declared. These members are also accessible from the other classes available in the same package.
- **default:** In Java, if no access specifier is used in the declaration of a class member then 'default' access specifier is considered as access specifier for that class member. A class member with 'default' access specifier can be accessible by the members of the same class and also accessible in the other classes available in the same package.

Example of a class definition in Java:

```
class Employee
{
    String emID, emName, emAddress;// default access specifier
    private String emContact, emDept, emDesig, emEmail;
    protected float basicSal, grossSal, emHRA, emDA;
```

```

public void readEmployeeInfo()
{
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter the Employee ID: ");
    emID = scan.next();
    scan.nextLine();
    System.out.println("Enter the Employee Name: ");
    emName = scan.next();
    scan.nextLine();
    System.out.println("Enter the Employee Address: ");
    emAddress = scan.next();
    scan.nextLine();
    System.out.println("Enter Employee Contact No.: ");
    emContact = scan.next();
    scan.nextLine();
    System.out.println("Enter Employee Department: ");
    emDept = scan.next();
    scan.nextLine();
    System.out.println("Enter Employee Designation: ");
    emDesig = scan.next();
    scan.nextLine();
    System.out.println("Enter Employee E-mail ID: ");
    emEmail = scan.next();
    scan.nextLine();
    System.out.println("Enter Employee Basic Salary: ");
    basicSal = scan.nextFloat();
    scan.nextLine();
    System.out.println("Enter Employee House Rent
Allowance(in Percentage): ");

    emHRA = scan.nextFloat();
    scan.nextLine();
    System.out.println("Enter Employee Dearness
Allowance(in Percentage): ");

    emDA = scan.nextFloat();
    scan.nextLine();
    scan.close();
}

public void displayEmployeeInfo()

```

```

    {
grossSal=basicSal+(emHRA/100)*basicSal+(emDA/100)*basicSal;

System.out.println("Employee ID:" +emID);
System.out.println("Employee Name:" +emName);
System.out.println("Employee Address:" +emAddress);
System.out.println("Employee Contact No.:" +emContact);
System.out.println("Employee Department:" +emDept);
System.out.println("Employee Designation:" +emDesig);
System.out.println("Employee E-mail ID:" +emEmail);
System.out.println("Employee Gross Salary:" +grossSal);

    }
}

```

In C++, members of a class are declared under any of the three access specifiers as presented in the following points.

- **public:** Class members declared under ‘public’ access specifier are accessible from any part of the program.
- **private:** Class members declared under ‘private’ access specifier are accessible only within the class where they are declared. These members cannot be accessible directly from outside the class where they are declared. In C++, if no access specifier is used in the declaration of a class member then by default that class member will be considered under ‘private’ access specifier.
- **protected:** Class members declared under ‘protected’ access specifier are accessible within the same class and in the derived classes of the class where they are declared.

Example of a class definition in C++:

```

class Employee
{

private:
char emID[10], emName[200], emAddress[300];
char emCont[10],emDep[50], emDsg[50], email[200];

protected:

```



```

float basicSal, grossSal,emHRA,emDA;

public:

void readEmployeeInfo()
{
    cout<<"\nEnter the Employee ID: ";
    cin>>emID;
    cout<<"\nEnter the Employee Name: ";
    cin>>emName;
    cout<<"\nEnter the Employee Address: ";
    cin>>emAddress;
    cout<<"\nEnter Employee Contact No.: ";
    cin>>emCont;
    cout<<"\nEnter Employee Department: ";
    cin>>emDep;
    cout<<"\nEnter Employee Designation: ";
    cin>>emDsg;
    cout<<"\nEnter Employee E-mail ID: ";
    cin>>email;
    cout<<"\nEnter Employee Basic Salary: ";
    cin>>basicSal;
    cout<<"\nEnter Employee House Rent Allowance(in Percentage): ";

    cin>>emHRA;
    cout<<"\nEnter Employee Dearness Allowance(in Percentage): ";
    cin>>emDA ;

}

void displayEmployeeInfo()
{
    grossSal=basicSal+(emHRA/100)*basicSal+(emDA/100)*basicSal;
    cout<<"\nEmployee ID:" <<emID;
    cout<<"\nEmployee Name:" <<emName;
    cout<<"\nEmployee Address:"<<emAddress;
    cout<<"\nEmployee Contact No.:"<<emCont;
    cout<<"\nEmployee Department:"<<emDep;
    cout<<"\nEmployee Designation:"<<emDsg;
    cout<<"\nEmployee E-mail ID:"<<email;
    cout<<"\nEmployee Gross Salary:"<<grossSal;

```

```
    }  
};
```

1.5 OBJECT

We have already learnt that Object Oriented Programming is based on the concept of object where object is an entity which contains data and functions. Objects can be created by instantiating classes.

Three basic characteristics of object are Identity, State and Behavior. Each object of a class must have a unique identity. It means that each object of any class must possess unique object name. The State of an object is represented by its data or properties. For example, an object of a class, 'student' may contain properties like name of the student, percentage of the student, course of the student, roll number of the student etc. The Behavior of an object is represented by the methods available in that object.

The general syntax to create object in Java is presented as follows.

```
Class_NameObject_Name = new Class_Name();
```

In the above syntax, 'Class_Name' is the name of the class, 'Object_Name' is the name of the object that is instantiated, 'new' is a keyword in Java used as Java operator for instantiation of objects and 'Class_Name()' is the constructor of the class.

For example, the following Java statement can be used to create object of the class, 'Employee'.

```
Employee emp1 = new Employee();
```

In the above statement, 'emp1' is the object that is instantiated.

The general syntax to create object in C++ is presented as follows.

```
Class_NameObject_Name;
```

For example, the following C++ statement can be used to create object of the class, 'Employee'.

```
Employee emp1;
```

Object name is used with the dot operator (.) to access the data or methods available in that object. We already have learnt that outside class, only public members of an object can be accessed directly with the object name. For example, if we want to access the function, 'void readEmployeeInfo()' defined in the class, 'Employee' then the following Java Statement can be used.

```
Employee emp1 = new Employee();  
emp1.readEmployeeInfo();
```

1.6 CONSTRUCTOR

A constructor is a special member function of a class used to initialize the objects of that class. Initialization of an object refers the process to set initial values for the data members of that object. When an object of a class is instantiated then an appropriate constructor available in that class will be invoked automatically. The name of a constructor must be same with name of the class where it is defined. Another important point regarding constructor is that no return type is provided in a constructor. In case of Inheritance, when a derived class object is instantiated then the constructor of the base class will be invoked first and then the derived class constructor will be invoked. Another important point regarding constructor is that constructor of a class cannot be declared as virtual.

Let us consider the following Java program to understand the concept of constructor.

Program 1.1: Java program to demonstrate the concept of Constructor.

```
import java.util.Scanner;  
class Employee  
{  
    private String emID, emName, emAddress, emContact;  
    private String emDept, emDesig, emDOB, emEmail;  
    protected double basicSal, grossSal, emHRA, emDA;
```

```

public void readEmployeeInfo()
{
    Scanner scan = new Scanner(System.in);

    System.out.println("Enter the Employee ID: ");
    emID = scan.next();
    scan.nextLine();
    System.out.println("Enter the Employee Name: ");
    emName = scan.nextLine();
    System.out.println("Enter the Employee Address: ");
    emAddress = scan.nextLine();
    System.out.println("Enter the Employee Contact No.: ");
    emContact = scan.next();
    scan.nextLine();
    System.out.println("Enter the Employee Department: ");
    emDept = scan.nextLine();
    System.out.println("Enter the Employee Designation: ");
    emDesig = scan.nextLine();
    System.out.println("Enter the Employee Date of Birth: ");
    emDOB = scan.next();
    scan.nextLine();
    System.out.println("Enter the Employee E-mail ID: ");
    emEmail = scan.next();
    scan.nextLine();
    scan.close();
}

public void displayEmployeeInfo()
{
    grossSal=basicSal + (emHRA/100)*basicSal +(emDA/100)*basicSal;
    System.out.println("Employee Information::");
    System.out.println("Employee ID:" +emID);
    System.out.println("Employee Name:" +emName);
    System.out.println("Employee Address:" +emAddress);
    System.out.println("Employee Contact No.:" +emContact);
    System.out.println("Employee Department:" +emDept);
    System.out.println("Employee Designation:" +emDesig);
    System.out.println("Employee Date of Birth:" +emDOB);
    System.out.println("Employee E-mail ID:" +emEmail);
    System.out.println("Employee Basic Salary:" +basicSal);
    System.out.println("Employee HRA Allowance:" +emHRA);
}

```

```

        System.out.println("Employee Dearness Allowance:" +emDA);
        System.out.println("Employee Gross Salary:" +grossSal);
    }

```

// Constructor with no parameter or Default Constructor

```

Employee()

```

```

{
    basicSal=50000.00;
    emHRA=10.00;
    emDA=53.00;
}

```

// Constructor with two double type parameters

```

Employee(double bas,double da) {
    basicSal=bas;
    emHRA=10.00;
    emDA=da;
}

```

//Constructor with three double type parameters

```

Employee(double bas,double hra, double da) {
    basicSal=bas;
    emHRA=hra;
    emDA=da;
}

```

// Copy Constructor

```

Employee(Employee emTemp)
{
    emID=emTemp.emID;
    emName=emTemp.emName;
    emAddress=emTemp.emAddress;
    emContact=emTemp.emContact;
    emDept=emTemp.emDept;
    emDesig=emTemp.emDesig;
    emDOB=emTemp.emDOB;
    emEmail=emTemp.emEmail;
    basicSal = emTemp.basicSal;
    emHRA = emTemp.emHRA;
    emDA = emTemp.emDA;
}

```

```

    }

}

class EmployeeInfo
{
private double bas,hra,da;
public static void main(String[] args)
    {
double bas,hra,da;
bas= 65000.00;
hra= 12.00;
da= 60.00;
        Employee emp=new Employee(bas,hra,da); /*Constructor with three double type parameters is invoked*/
emp.readEmployeeInfo();
emp.displayEmployeeInfo();

Employee empCopy= new Employee(emp);/*Copy constructor is invoked*/
empCopy.displayEmployeeInfo();

    }
}

```

Output of the Program:

```

Enter the Employee ID: E0012
Enter the Employee Name:PrantikDeka
Enter the Employee Address: Guwahati, Assam
Enter the Employee Contact No.: 9999999999
Enter the Employee Department: HR
Enter the Employee Designation: Manager
Enter the Employee Date of Birth: 12-09-1991
Enter the Employee E-mail ID: pr@dd.com

```

```

Employee Information::
Employee ID:E0012
Employee Name:PrantikDeka
Employee Address:Guwahati, Assam
Employee Contact No.:9999999999
Employee Department:HR
Employee Designation:Manager

```

Employee Date of Birth:12-09-1991
Employee E-mail ID:pr@dd.com
Employee Basic Salary:65000.0
Employee HRA Allowance:12.0
Employee Dearness Allowance:60.0
Employee Gross Salary:111800.0

Employee Information::
Employee ID:E0012
Employee Name:PrantikDeka
Employee Address:Guwahati, Assam
Employee Contact No.:9999999999
Employee Department:HR
Employee Designation:Manager
Employee Date of Birth:12-09-1991
Employee E-mail ID:pr@dd.com
Employee Basic Salary:65000.0
Employee HRA Allowance:12.0
Employee Dearness Allowance:60.0
Employee Gross Salary:111800.0

In Program 1.1, it is observed that three constructors are defined in the class, 'Employee'. So, multiple constructors can be defined in the same class but they must be different from each other in terms of their number of parameters or type of corresponding parameters or both. It means two constructors in the same class with same number of parameters and same data types of the corresponding parameters are not allowed. Defining more than one constructor in the same class is termed as **Constructor Overloading**. If one or more than one parameters are passed into a constructor then that constructor is termed as **Parameterized constructor**. In Program 1.1, 'Employee(double bas,double hra, double da)' is an example of parameterized constructor.

It may be possible that a class is defined without defining a constructor explicitly in it. In that case, programming languages like Java and C++ deliver a system generated constructor for that class. Such constructor is termed as **Default constructor**. The default constructor does not have any parameter. This constructor initializes the objects by setting default values to the different data members of the corresponding objects. The default values are dependent upon the

type of the data member and the corresponding programming language. For example in Java, the default value for an 'int' type data member is 0, for a 'boolean' type data member is 'false' etc. It may be also possible that a constructor is defined for a class where the constructor does not have any parameter. Such a constructor is also termed as default constructor. Through this user-defined default constructor, the programmer can initialize the object by setting default values for the data members of the corresponding objects as per the requirements in the corresponding program. In Program 1.1, a default constructor is defined where the default values for 'basicSal', 'emHRA' and 'emDA' are 50000.00, 10.00 and 53.00 respectively.

When a constructor is defined in a class to initialize an object with the values of the data members of an existing object then such a constructor is termed as **Copy constructor**. Copy constructor is used when it is required to instantiate an object with the same state of an old object. In Program 1.1, a Copy constructor ('Employee(Employee emTemp)') is defined where an existing object is passed as parameter into the constructor. From the output of the program, it is observed that different values to the properties of the object instantiated by using the Copy constructor are same with the values to the corresponding properties of the object that is passed as parameter into the Copy constructor.

Program 1.2: C++ program to demonstrate the concept of Constructor

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

```
class Employee
{
private:
char emID[10], emName[200], emAdd[200], emCont[200];
char emDep[50], emDes[100], emDOB[12], emEmail[100];
float basicSal, grossSal, emHRA, emDA;
```


public:

void readEmployeeInfo()

```
{
    cout<<"\nEnter the Employee ID: ";
    cin>>emID;
    cout<<"\nEnter the Employee Name: ";
    gets(emName);
    cout<<"\nEnter the Employee Address: ";
    gets(emAdd);
    cout<<"\nEnter the Employee Contact No.: ";
    cin>>emCont;
    cout<<"\nEnter the Employee Department: ";
    gets(emDep);
    cout<<"\nEnter the Employee Designation: ";
    gets(emDes);
    cout<<"\nEnter the Employee Date of Birth: ";
    cin>>emDOB;
    cout<<"\nEnter the Employee E-mail ID: ";
    cin>>emEmail;
}
```

void displayEmployeeInfo()

```
{
    grossSal=basicSal      +      (emHRA/100)*basicSal
    +(emDA/100)*basicSal;
```

```
cout<<"\nEmployee Information: ";
    cout<<"\nEmployee ID:"<<emID;
    cout<<"\nEmployee Name:"<<emName;
    cout<<"\nEmployee Address:"<<emAdd;
    cout<<"\nEmployee Contact No.:"<<emCont;
    cout<<"\nEmployee Department:"<<emDep;
    cout<<"\nEmployee Designation:"<<emDes;
    cout<<"\nEmployee Date of Birth:"<<emDOB;
    cout<<"\nEmployee E-mail ID:"<<emEmail;
    cout<<"\nEmployee Basic Salary:"<<basicSal;
    cout<<"\nEmployee HRA Allowance:"<<emHRA;
    cout<<"\nEmployee Dearness Allowance:"<<emDA;
    cout<<"\nEmployee Gross Salary:"<<grossSal;
```

```

    }

    Employee()    // Constructor with no parameter
    {
        basicSal=50000.00;
        emHRA=10.00;
        emDA=53.00;
    }

    Employee(float bas, float da) /* Constructor with two float
type parameters*/
    {
        basicSal=bas;
        emHRA=10.00;
        emDA=da;
    }

    Employee(float bas, float hra, float da) /*Constructor with
three float type parameters*/
    {
        basicSal=bas;
        emHRA=hra;
        emDA=da;
    }

    Employee(Employee &emTemp)// Copy Constructor
    {
        strcpy(emID,emTemp.emID);
        strcpy(emName,emTemp.emName);
        strcpy(emAdd,emTemp.emAdd);
        strcpy(emCont,emTemp.emCont);
        strcpy(emDep,emTemp.emDep);
        strcpy(emDes,emTemp.emDes);
        strcpy(emDOB,emTemp.emDOB);
        strcpy(emEmail,emTemp.emEmail);
        basicSal = emTemp.basicSal;
        emHRA = emTemp.emHRA;
        emDA = emTemp.emDA;
    }

```

```

} ;

int main()
{
float bas,hra,da;
clrscr();
bas= 65000.00;
hra= 12.00;
da= 60.00;
    Employee emp(bas,hra,da); /*Constructor with three float type
parameters is invoked*/

emp.readEmployeeInfo();
emp.displayEmployeeInfo();
    Employee empCopy(emp); //Copy constructor is invoked
empCopy.displayEmployeeInfo();
getch();
return(0);

}

```

1.7 DESTRUCTOR

A destructor is also a special member function of a class. It is invoked automatically when an object is no longer required. The job of a destructor is to release the resources allocated to the corresponding object. A class can contain only one destructor. It means that destructor overloading is not possible. A destructor does not have any parameter and return type. In C++, the name of a destructor is same with the name of the corresponding class preceded by the character, tilde (~). For example, if the name of the class is 'className' then the destructor of the class will be:

```

~className{
    // Body of the Destructor
}

```

In case of C++, if a class is defined without defining a destructor in it then the compiler will generate a default destructor for that class.

In case of Inheritance, when an object of a derived class is no longer needed and it is destroyed then the destructor of the derived class will be invoked first and then the base class destructor will be invoked. In C++, destructor of a base class can be declared as virtual. In that case, if a base class pointer point to an object of a derived class and that derived class object is destroyed then the destructor of the derived class and destructors of all its base classes are called. On the other hand, if the destructor of the base class is not declared as virtual and the object of the derived class pointed to by a base class pointer, is destroyed then only the destructor of the base class is called.

1.8 TEMPLATES

Template is a useful feature provided in C++ to allow developing reusable functions and classes that can be used as single frameworks for supporting different data types. It allows the developer to write programs in such a way that the program will work on different data types without writing same code for each data type. So, use of templates can improve the code reusability, maintainability and flexibility in C++ programming.

In general, two types of templates can be declared in C++ that are **Function template** and **Class template**.

Function template: A Function template is a template which is declared for function. A Function template allows writing a function which can perform its job by supporting different data types available in C++. It means that if we declare a function template then we are not required to write multiple copies of the same function for different types of parameters or data. When a function template is declared and a function call correspond to that template is performed then the compiler generates a new function using that function template depending upon the type of the parameters in the function call. Then this compiler generated function is invoked. The function generated by compiler using a function template is termed as **template function**.

The syntax for declaring a Function template is presented as follows.

```

template<class D,.....>
Return_Type functionName( Parameters)
{
    // Statements of the template function
}

```

In the above syntax, '*template*' is a keyword in C++ and 'D' is a template data type. A function template can be declared with more than one template data types. The parameter list of the template function must contain at least one template type parameter. Let us consider the following C++ program to understand the concept of Function template.

Program 1.3: C++ program to demonstrate the concept of Function template.

```

#include<iostream.h>
#include<conio.h>

// Function templates

template<class D>

D average(D nm1,D nm2,D nm3)
{
    return((nm1+nm2+nm3)/3);
}

template<class D>

D average(D aN[],int n)
{
    int i;
    D temp=0;
    for(i=0;i<n;i++)
    {
        temp+=aN[i];
    }
    return( temp/n);
}

```

```

    }

int main()
{
    int nm1,nm2,nm3,av,i,n;
    float arrN[100];
    float rnm1,rnm2,rnm3,rav;
    clrscr();
    cout<<"\n Enter three Integer numbers::";
    cin>>nm1>>nm2>>nm3;
    av=average(nm1,nm2,nm3);
    cout<<"\n Average of the three Integer numbers is="<<av;

    cout<<"\n\n Enter three Real numbers::";
    cin>>rnm1>>rnm2>>rnm3;
    rav=average(rnm1,rnm2,rnm3);
    cout<<"\n Average of the three Real numbers is="<<rav;

    cout<<"\n\n Enter the total number of data to be stored in the array=";
    cin>>n;
    if(n>100)
        cout<<"\n Wrong input";
    else
    {
        cout<<"\n Enter "<<n<<" number of real numbers into the array::";
        for(i=0;i<n;i++)
        {
            cout<<"\nEnter "<<i+1<<"th data into the array=";
            cin>>arrN[i];
        }
        rav=average(arrN,n);
        cout<<"\n Average of the numbers stored in the array is="<<rav;
    }
    getch();
    return(0);
}

```

Output of the program:

```

Enter three Integer numbers:: 23 45 67
Average of the three Integer numbers is= 45

```

Enter three Real numbers:: 12.6 56.8 78.12
Average of the three Real numbers is= 49.173332

Enter the total number of data to be stored in the array= 2
Enter 2 number of real numbers into the array::
Enter 1th data into the array= 45.55
Enter 2th data into the array= 65.23

Average of the numbers stored in the array is= 55.389999

In **Program 1.3**, a function template is declared to calculate the average of three values. From the output of the program it is observed that when three integer values are passed as parameters to the template function then it returns the average of these numbers that is also an integer number. Again when three real numbers are passed as parameters then it returns the average which is also a real number. Hence, the template function works on both 'int' and 'float' type data.

In **Program 1.3**, it is observed that two function templates are declared with same function name where the first template function has three template type parameters and the second template function has two parameters where one is a template type array and the other one is an 'int' type parameter. This type of multiple declarations of Function templates is referred as overloading of function templates.

In **Program 1.3**, it is also observed that the declared function template can handle a single template data type. But functions may require more than one parameter of different data types so that they can perform their jobs. In such situation, the function template is declared with multiple template data types as per requirement. For example, let us consider the following C++ program.

Program 1.4: C++ program to demonstrate the use of Function template with more than one template data types.

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

```
// Function template with two template data types
```

```
template<class D, class E>
```

```

void displayData(D data1,E data2)
{
cout<<"\n The first data="<<data1;
cout<<"\n The second data="<<data2;
}

int main()
{
int intData;
float floatData;
char strData[50];
clrscr();
cout<<"\n Enter an integer number=";
cin>>intData;

cout<<"\n\n Enter a Real number=";
cin>>floatData;

cout<<"\n Entered data are:";
displayData(intData,floatData);
getch();
return(0);
}

```

Output of the program:

```

Enter an integer number= 12
Enter a Real number= 34.89

```

Entered data are:

```

The first data= 12
The second data= 34.89

```

In **Program 1.4**, a function template is declared with two template data types ('D' and 'E').

Class template: The template declared for class is termed as class template. A class template allows creating a class that can operate on multiple data types. It means that class templates allow creating

classes for performing same tasks for multiple data types without writing same codes for each of the data types.

The syntax of declaring Class templates is presented as follows.

```
template< class D1, class D2, .....>
class Class_Name
{
private:
    // D1 type data member
    D1 data1;

public:
    //Member function with a D2 type parameter
    void function1(D2 data2);
};
```

In the above syntax, declaration of one data member and one member function is provided in the class template.

Let us consider the following C++ program to understand the concept of Class templates.

Program 1.5: C++ program to demonstrate the concept of Class template.

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

// Class template
template<class D>
class Sort
{
private:
    D arr[50],temp;
    int i,j;
public:
    void readData(int nData)
    {
```

```

        cout<<"\n Enter "<<nData<<" number of data into the
array:.";
        for(i=0;i<nData;i++)
        {
            cout<<"\nEnter "<<i+1<<"th data into the array=";
            cin>>arr[i];
        }
    }
    void displayData(int nData)
    {
        cout<<"\n Data available in the array are::\n";
        for(i=0;i<nData;i++)
        {
            cout<<"\t"<<arr[i];
        }
    }

    void bubble_sort(int nData)
    {
        for (int i = 0; i <nData - 1; i++)
        {
            for (int j = 0; j <nData - i - 1; j++)
            {
                if (arr[j] >arr[j + 1])
                {
                    temp= arr[j];
                    arr[j]=arr[j+1];
                    arr[j+1]=temp;
                }
            }
        }
    }
};

int main()
{
    int N;
    Sort <int> obj1;
    Sort <float> obj2;

```

```

clrscr();
cout<<"\n\n Enter the total number of data to be stored in the array=";
cin>>N;
if(N>50)
    cout<<"\n Wrong input";
else
{
    cout<<"\n Sorting for Integer numbers::";
    obj1.readData(N);
    cout<<"\n Before sorting, the data in the array are::";
    obj1.displayData(N);
    obj1.bubble_sort(N);
    cout<<"\n After sorting, the data in the array are::";
    obj1.displayData(N);

    cout<<"\n Sorting for Real numbers::";
    obj2.readData(N);
    cout<<"\n Before sorting, the data in the array are::";
    obj2.displayData(N);
    obj2.bubble_sort(N);
    cout<<"\n After sorting, the data in the array are::";
    obj2.displayData(N);
}
getch();
return 0;

}

```

Output of the program:

Enter the total number of data to be stored in the array= 4

Sorting for Integer numbers::

Enter 4 number of data into the array::

Enter 1th data into the array= 23

Enter 2th data into the array= 4

Enter 3th data into the array= 56

Enter 4th data into the array= 8

Before sorting, the data in the array are::

Data available in the array are::

23 4 56 8

After sorting, the data in the array are::

Data available in the array are::

4 8 23 56

Sorting for Real numbers::

Enter 4 number of data into the array::

Enter 1th data into the array= 12.5

Enter 2th data into the array= 12.1

Enter 3th data into the array= 56.3

Enter 4th data into the array= 5.9

Before sorting, the data in the array are::

12.5 12.1 56.3 5.9

After sorting, the data in the array are::

5.9 12.1 12.5 56.3

In **Program 1.5**, a class template is declared with one template data type. From the output of the program, it is observed that using the class template, sorting of data using Bubble sort algorithm can be performed on both integer and real numbers stored in an integer array and a float type array respectively.

Check Your Progress

1. Choose the correct option
 - (a) Which of the following is not a main property of Object Oriented Programming (OOP)?
 - (i) Abstraction
 - (ii) Encapsulation
 - (iii) Template
 - (iv) Inheritance

- (b) Which of the following is one of the advantages of OOP?
- (i) Code reusability is degraded
 - (ii) Maintenance becomes simpler
 - (iii) Adding a new feature become complex
 - (iv) None of the above
- (c) _____ permits combining data and methods together to form a single unit.
- (i) Encapsulation
 - (ii) Abstraction
 - (iii) Polymorphism
 - (iv) None of the above
- (d) Which of the following is not an access specifier in C++?
- (i) public
 - (ii) private
 - (iii) protected
 - (iv) default
- (e) Class members declared with the access specifier, _____ are accessible by only the members of the class where they are declared.
- (i) public
 - (ii) private
 - (iii) protected
 - (iv) None of the above
- (f) Which of the following access specifier is associated with Inheritance?
- (i) public
 - (ii) private
 - (iii) protected
 - (iv) None of the above
- (g) Which of the following is a basic characteristic of objects?
- (i) Identity
 - (ii) State
 - (iii) Behavior
 - (iv) All of the above

- (h) Find out the false statement regarding constructor.
- (i) Multiple constructors can be defined in a single class.
 - (ii) Constructor can be declared as virtual.
 - (iii) Constructor is invoked to initialize the objects.
 - (iv) All of the above
- (i) _____ constructor does not have any parameter.
- (i) Copy constructor
 - (ii) Parameterized constructor
 - (iii) Default constructor
 - (iv) None of the above
- (j) Find out the True statement regarding destructor.
- (ii) Destructor can be declared as virtual.
 - (iii) Destructor is invoked when an object is instantiated.
 - (iv) Initialization of object is performed by the destructor
- (k) Which of the following is not a type of template in C++?
- (i) Class template
 - (ii) Object template
 - (iii) Function template
 - (iv) None of the above

1.9 SUMMING UP

- Object-Oriented Programming (OOP) is a programming paradigm that gives more emphasis on data than the algorithm.
- Improved code reusability, simpler maintenance, improved modularity, improved data security and integrity and easier debugging and testing are some of the general advantages of OOP.
- Four main properties of OOP are Data Abstraction, Encapsulation, Inheritance and Polymorphism.

- Data abstraction in OOP allows hiding the implementation particulars of objects and offering only the necessary features to the users.
- Data abstraction in OOP is implemented through Encapsulation. Encapsulation in OOP permits combining data and methods together to form a single unit.
- Encapsulation is implemented by defining classes in OOP.
- A class is a model or a blueprint that forms a single unit by enclosing data and methods.
- A nested class is a class that is defined inside a class.
- In OOP, access specifiers are associated with the members of classes so that visibility and accessibility of the members of a class from outside the class can be controlled.
- Four access specifiers available in Java are public, private, protected and default.
- Three access specifiers available in C++ are public, private, protected.
- Class members declared with the access specifier, 'public' are accessible from any part of the program.
- Class members declared with the access specifier, 'private' are accessible only inside the class where they are declared.
- 'protected' access specifier is associated with Inheritance. Class members declared with the access specifier, 'protected' are accessible by the same class members and by the members of the derived classes of the class where they are declared.
- In Java, if no access specifier is used in the declaration of a class member then 'default' access specifier is considered as access specifier for that class member. A class member with 'default' access specifier can be accessible by the members of the same class and also accessible in the other classes available in the same package. In C++, by default, the access specifier of a class member is private.
- Object Oriented Programming is based on the concept of object where object is an entity which contains data and functions. An object is actually a variable of a class.
- Three basic characteristics of object are Identity, State and Behavior.
- Object name is used with the dot operator (.) to access the data or methods available in that object.

- A constructor is a special member function of a class used to initialize the objects of that class. The name of a constructor must be same with name of the class where it is defined. Constructor does not have any return type.
- More than one constructor can be defined in a single class and it is termed as constructor overloading.
- If one or more than one parameters are passed into a constructor then that constructor is termed as Parameterized constructor.
- If no constructor is defined in a class then programming languages like Java and C++ deliver a system generated constructor for that class. Such constructor is termed as Default constructor. The default constructor does not have any parameter.
- When a constructor is defined in a class to initialize an object with the values of the data members of an existing object then such a constructor is termed as Copy constructor.
- A destructor is a special member function of a class and it is invoked automatically when an object is destroyed. The job of a destructor is to release the resources allocated to the corresponding object. A class can contain only one destructor. A destructor does not have any parameter and return type. In C++, the name of a destructor is same with the class name preceded by the character, tilde (~).
- In C++, Template allows developing reusable functions and classes that can be used as single frameworks for supporting different data types.
- Use of templates can improve the code reusability, maintainability and flexibility in C++ programming.
- In general, two types of templates can be declared in C++ that are Function template and Class template. A Function template is a template which is declared for function. A Function template allows writing a function which can perform its job by supporting different data types available in C++.
- The template declared for class is termed as class template.

ANSWERS TO CHECK YOUR PROGRESS

1.

- (a) (iii) Template
- (b) (ii) Maintenance becomes simpler
- (c) (i) Encapsulation
- (d) (iv) default
- (e) (ii) private
- (f) (iii) protected
- (g) (iv) All of the above
- (h) (ii) Constructor can be declared as virtual.
- (i) (iii) Default constructor
- (j) (ii) Destructor can be declared as virtual.
- (k) (ii) Object template

1.10 POSSIBLE QUESTIONS

1. Write down the advantages of Object Oriented Programming (OOP).
2. Define class and object.
3. Explain different access specifiers that are available in Java and C++.
4. What is Constructor overloading? Give example.
5. What is Constructor? Explain different types of Constructors.
6. What is destructor? Write down the differences between Constructor and Destructor.
7. Write down a short note on Template.

1.11 REFERENCES AND SUGGESTED READINGS

- 1) Venugopal, K. R., Rajkumar, Ravishankar, T. *Mastering C++*. Tata McGraw-Hill Education, 2001.
- 2) Balagurusamy, E. *Object Oriented Programming with C++*. Tata McGraw-Hill, 2006
- 3) Jana, Debasish. *Java and object-oriented programming paradigm*. PHI Learning Pvt. Ltd., 2005.
- 4) Schildt, Herbert. *Java: the complete reference*. McGraw-Hill Education Group, 2014.

---x---

UNIT 2: INHERITANCE

Unit Structure:

- 2.1 Introduction
- 2.2 Objectives
- 2.3 Introduction to Inheritance
- 2.4 Advantages of Inheritance
- 2.5 Casting Up the Hierarchy
- 2.6 Types of Inheritance
 - 2.6.1 Single Inheritance
 - 2.6.2 Multiple Inheritance
 - 2.6.3 Hierarchical Inheritance
 - 2.6.4 Multi-Level Inheritance
 - 2.6.5 Hybrid Inheritance
- 2.7 Virtual Base Class
- 2.8 Summing Up
- 2.9 Possible Questions
- 2.10 References and Suggested Readings

2.1 INTRODUCTION

In the earlier unit, we have learnt about two important properties of Object Oriented Programming (OOP) that are Abstraction and Encapsulation. In this unit, we are going to discuss about Inheritance which is also an important property of OOP. Inheritance allows defining new sub classes or child classes from already existing classes. Advantages of Inheritance and different types of Inheritance are going to be discussed in this unit. Concept of Virtual Base class in C++ will also be discussed here.

2.2 OBJECTIVES

After reading this unit, you are expected to be able to learn:

- What is Inheritance?
- About the advantages of Inheritance.

- About casting up the hierarchy.
- About different types of Inheritance in Object Oriented Programming (OOP).
- About the concept of Virtual Base class in C++.

2.3 INTRODUCTION TO INHERITANCE

We have already learnt that Inheritance is one of the main properties of OOP. Inheritance is a technique provided in OOP to create new sub or child classes from the already existing classes. In this process, the already existing classes are called as base class. A base class can also be referred as parent class or super class. On the other hand, the newly created classes from a base class are called as derived classes. A derived class can also be referred as sub class or child class. A derived class inherits the properties and behaviors of its base class and it may also contain its own properties and behaviors. Role of access specifiers in OOP is already discussed in the earlier unit. In this regard, one important point is that private members of a base class are not inherited to its derived classes. Public and protected members of a base class can be inherited to its derived classes. In Java, the members of a base class with 'default' access specifier can be inherited to its derived classes that are available in the same package. But derived classes from different package cannot inherit them.

The syntax to implement Inheritance in Java is presented as follows.

```
class Base-Class{  
    //Body of the Base class  
}  
  
class Derived-Class extends Base-Class{  
    // Body of the Derived class  
}
```

In the above syntax, 'Derived-Class' is a child class derived from the base class, 'Base-Class' where '*extends*' is a keyword in Java.

Example:

```
class UniversityPeople
{

    public void displayInfo()
    {
        System.out.println("Class for University People Information::");
    }
}

class Student extends UniversityPeople // Child class creation
{

    public void displayStudentInfo()
    {
        System.out.println("Class for Student Information Derived from
the Base class, UniversityPeople");
    }
}
```

In the above example, ‘UniversityPeople’ is the base class and ‘Student’ is the derived class extended from ‘UniversityPeople’.

The syntax to implement Inheritance in C++ is presented as follows.

```
class Base-Class{
    //Body of the Base class
};

class Derived-Class : [Visibility-Mode] Base-Class{
    // Body of the Derived class
};
```

In the above syntax, ‘Derived-Class’ is a child class derived from the base class, ‘Base-Class’. Here, Visibility-Mode may be used to state how the features of the base class are inherited to its derived class. ‘public’ or ‘private’ or ‘protected’ can be used as Visibility-Mode in C++. If no Visibility-Mode is provided in the

definition of a derived class then by default the Visibility-Mode will be 'private'. Let us consider the following points related to the Visibility-Mode.

- If the Visibility-Mode is '**public**' then:
 - Public members of the base class will be inherited to the derived class as public members.
 - Protected members of the base class will be inherited to the derived class as protected members.
- If the Visibility-Mode is '**protected**' then:
 - Public members of the base class will be inherited to the derived class as protected members.
 - Protected members of the base class will be inherited to the derived class as protected members.
- If the Visibility-Mode is '**private**' then:
 - Public members of the base class will be inherited to the derived class as private members.
 - Protected members of the base class will be inherited to the derived class as private members.

Example:

```
class UniversityPeople
```

```
{  
  
    public void displayInfo()  
    {  
        cout<<"\nClass for University People Information::";  
    }  
}
```

```
class Student: public UniversityPeople // Child class creation
```

```
{  
  
    public void displayStudentInfo()  
    {  
        cout<<"\nClass for Student Information Derived from the Base  
class, UniversityPeople ";  
    }  
}
```

In the above example, 'UniversityPeople' is the base class and 'Student' is the child class derived from 'UniversityPeople' where the visibility mode of the inheritance is 'public'.

2.4 ADVANTAGES OF INHERITANCE

Inheritance is a very useful property of Object-Oriented Programming (OOP). Advantages of Inheritance in OOP are presented in the following points.

- New features can be easily included to an already existing application without changing any existing code of the application by implementing Inheritance.
- Reusability of already existing code can be increased through Inheritance. Base class properties and behaviors can be reused in the derived classes due to Inheritance.
- Due to Inheritance, redundancy in code is decreased as it allows reuse of existing code.
- Code maintenance and modification may also become easier due to Inheritance as any modification performed in a base class is also reflected in its derived classes.
- Run-time polymorphism is supported in OOP through Inheritance. A base class function can be overridden by a derived class function at run-time.
- Real-world relationships can be implemented to develop an application by implementing Inheritance in OOP.
- Development time of an application using OOP can be reduced by implementing Inheritance.

2.5 CASTING UP THE HIERARCHY

In Inheritance, casting up the hierarchy refers to use a base class pointer or a base class reference for pointing or referring a derived class object. But this base class pointer or reference variable can be used to access only the features of the base class. It can be used to override a base class function by a derived class function at run-time. Casting up the hierarchy is also termed as Upcasting. In

C++, function overriding using Upcasting is possible by applying the concept of virtual function. Details of virtual function will be discussed in the next unit. Let us consider the following C++ program to get a clear idea about Upcasting.

Program 2.1 C++ program to demonstrate Upcasting

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

class UniversityPeople
{
private:
    char upName[200], upAddress[300], upContact[10];
public:
    void inputInfo()
    {
        cout<<"Input People Information:.";
        cout<<"\nInput Name: ";
        gets(upName);
        cout<<"\nInput Address: ";
        gets(upAddress);
        cout<<"\nInput Contact Number: ";
        cin>>upContact;
    }
    void displayInfo()
    {
        cout<<"\nPeople Information:.";
        cout<<"\nName="<<upName;
        cout<<"\nAddress ="<<upAddress;
        cout<<"\nContact Number ="<<upContact;
    }
};

class Student : public UniversityPeople
{
private:
    char stCourse[100];
```

```

        int stSemester,stRollNo;

public:
    void inputInfo()
    {
        cout<<"\nInput Information of the Student::";
        cout<<"\nInput Student Course: ";
        gets(stCourse);
        cout<<"\nInput Semester: ";
        cin>>stSemester;
        cout<<"\nInput Student Roll Number: ";
        cin>>stRollNo;
    }

    void displayInfo()
    {
        cout<<"\nPeople Type-Student";
        cout<<"\nRoll Number="<<stRollNo;
        cout<<"\nCourse of the Student="<<stCourse;
        cout<<"\nSemester of the Student="<<stSemester;
    }
};

int main()
{
    UniversityPeople *up;
    Student st;
    clrscr();
    up=&st; //Upcasting
    up->inputInfo();
    up->displayInfo();
    getch();
    return(0);
}

```

Output of the program:

```

Input People Information::
Input Name: BhargabSarma
        Input Address: Nalbari, Assam
        Input Contact Number: 777777

```


People Information::

Name=BhargabSarma

Address=Nalbari, Assam

Contact Number =777777

In the above program, UniversityPeople is a base class and Student is its derived class. It is observed that 'up' is a base class pointer used to point the derived class object, 'st'. From the output, it is also observed that the base class methods are invoked when 'up' is used to call these methods.

2.6 TYPES OF INHERITANCE

In General, Object-Oriented Programming (OOP) supports five different types of Inheritance. These five types of Inheritance are:

- (a) Single Inheritance,
- (b) Multiple Inheritance,
- (c) Hierarchical Inheritance,
- (d) Multi-Level Inheritance,
- (e) Hybrid Inheritance.

2.6.1 Single Inheritance

When only one child class is derived from only one base class then it is termed as Single Inheritance. Let us consider the following Java program to get a clear idea about Single Inheritance.

Program 2.2: Java program to demonstrate Single Inheritance

```
import java.util.Scanner;

class UniversityPeople
{
    private String upName, upAddress, upContact;
    public void inputInfo(Scanner scan)
    {
        System.out.println("Input People Information::");
        System.out.println("Input Name: ");
        upName = scan.nextLine();
```

```

System.out.println("Input Address: ");
upAddress = scan.nextLine();
System.out.println("Input Contact Number: ");
upContact = scan.nextLine();
    }
public void displayInfo()
    {
System.out.println("People Information::");
System.out.println("Name="+upName);
System.out.println("Address =" +upAddress);
System.out.println("Contact Number =" +upContact);
    }
}
class Student extends UniversityPeople
{
private String stDept, stCourse;
private int stSemester,stRollNo;

public void inputStudentInfo(Scanner scan)
    {
System.out.println("Input Information of the Student::");
System.out.println("Input Student Department: ");
stDept = scan.nextLine();

System.out.println("Input Student Course: ");
stCourse = scan.nextLine();
System.out.println("Input Semester: ");

stSemester = scan.nextInt();
System.out.println("Input Student Roll Number: ");

stRollNo = scan.nextInt();

    }

public void displayStudentInfo()
    {
System.out.println("People Type-Student");
System.out.println("Roll Number="+stRollNo);
System.out.println("Department of the Student =" +stDept);

```

```
System.out.println("Course of the Student="+stCourse);
System.out.println("Semester of the Student="+stSemester);
```

```
    }
}

class UniversityInfo
{
public static void main(String[] args)
{
    Scanner scan = new Scanner(System.in);
    Student st1 = new Student();
    st1.inputInfo(scan);
    st1.inputStudentInfo(scan);
    st1.displayInfo();
    st1.displayStudentInfo();
}
}
```

Output of the program:

```
Input People Information::
Input Name:AmitSarma
Input Address: Guwahati, Assam
Input Contact Number: 22222
Input Information of the Student::
Input Student Department: GUCDOE
Input Student Course: M.A. in Assamese
Input Semester: 2
Input Student Roll Number: 25
```

```
People Information::
Name=AmitSarma
Address =Guwahati, Assam
Contact Number =22222
People Type-Student
Roll Number=25
Department of the Student =GUCDOE
Course of the Student=M.A. in Assamese
Semester of the Student=2
```

In the above program, 'Student' is the only derived class which is derived from the base class, 'UniversityPeople'. So, it is an example of Single Inheritance. The diagrammatic representation of this Single Inheritance is presented as follows.

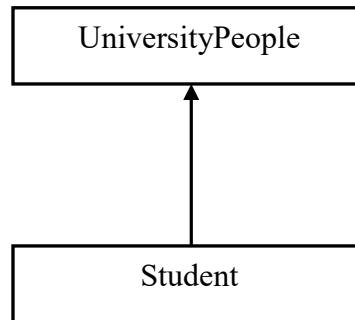


Figure 2.1: Diagrammatic Representation of a Single Inheritance

2.6.2 Multiple Inheritance

When a sub class is derived from more than one base class then it is termed as Multiple Inheritance. It means that a child class can have multiple parent classes in case of Multiple Inheritance. Now let us consider the following C++ program to get a clear idea about Multiple Inheritance.

Program 2.3 C++ program to demonstrate Multiple Inheritance

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

class UniversityPeople
{
private:
char upName[200], upAddress[300], upContact[10];
public:
void inputInfo()
{
cout<<"Input People Information::";
cout<<"\nInput Name: ";
```

```

    gets(upName);
    cout<<"\nInput Address: ";
    gets(upAddress);
    cout<<"\nInput Contact Number: ";
    cin>>upContact;
    }
void displayInfo()
{
    cout<<"\nPeople Information::";
    cout<<"\nName="<<upName;
    cout<<"\nAddress ="<<upAddress;
    cout<<"\nContact Number ="<<upContact;
}
};

class Department
{
private:
    char deptName[200],deptHOD[200],DOE[11];
public:
    void inputDepartmentInfo()
    {
        cout<<"\n Input Department Information::";
        cout<<"\n Input Department Name:";
        gets(deptName);
        cout<<"\n Input Head of the Department:";
        gets(deptHOD);
        cout<<"\n Input Date of Establishment:";
        cin>>DOE;
    }

    void displayDepartmentInfo()
    {
        cout<<"\n Department Name=";
        cout<<deptName;
        cout<<"\n Head of the Department=";
        cout<<deptHOD;
        cout<<"\n Date of Establishment=";
        cout<<DOE;
    }
};

```

```

class Student : public UniversityPeople, public Department
{
private:
char stCourse[100];
int stSemester,stRollNo;

public:
void inputStudentInfo()
{
cout<<"\nInput Information of the Student:.";
cout<<"\nInput Student Course: ";
gets(stCourse);
cout<<"\nInput Semester: ";
cin>>stSemester;
cout<<"\nInput Student Roll Number: ";
cin>>stRollNo;
}

void displayStudentInfo()
{
cout<<"\nPeople Type-Student";
cout<<"\nRoll Number="<<stRollNo;
cout<<"\nCourse of the Student="<<stCourse;
cout<<"\nSemester of the Student="<<stSemester;
}
};

int main()
{
Student st;
clrscr();
st.inputInfo();
st.inputDepartmentInfo();
st.inputStudentInfo();
st.displayInfo();
st.displayDepartmentInfo();
st.displayStudentInfo();
getch();
return(0);
}

```

Output of the program:

Input People Information::

Input Name:MridulGogoi

Input Address:Dibrugarh, Assam

Input Contact Number: 9999

Input Department Information::

Input Department Name: Assamese

Input Head of the Department: Prof. BimalBarua

Input Date of Establishment: 03-07-1991

Input Information of the Student::

Input Student Course: M.A. in Assamese

Input Semester:3

Input Student Roll Number: 7

People Information::

Name=MridulGogoi

Address =Dibrugarh, Assam

Contact Number =9999

Department Name=Assamese

Head of the Department=Prof. BimalBarua

Date of Establishment=03-07-1991

People Type-Student

Roll Number=7

Course of the Student=M.A. in Assamese

Semester of the Student=3

In the above program, it is observed that 'Student' is a derived class and it is derived from two base classes that are 'UniversityPeople' and 'Department'. The diagrammatic representation of this Multiple Inheritance is presented as follows.

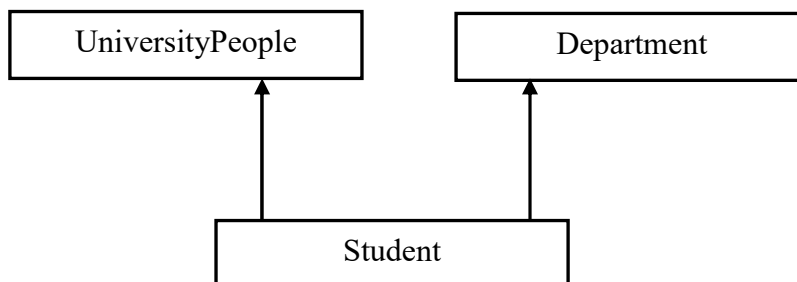


Figure 2.2: Diagrammatic Representation of a Multiple Inheritance

STOP TO CONSIDER

Multiple Inheritance is not supported in Java Programming.

2.6.3 Hierarchical Inheritance

Hierarchical Inheritance is the opposite of Multiple Inheritance. When more than one sub classes are derived from a single base class then it is termed as Hierarchical Inheritance. Let us consider the following Java program to understand Hierarchical Inheritance.

Program 2.4: Java program to demonstrate Hierarchical Inheritance

```
import java.util.Scanner;

class UniversityPeople
{
    private String upName, upAddress, upContact;
    public void inputInfo(Scanner scan)
    {
        System.out.println("Input People Information::");
        System.out.println("Input Name: ");
        upName = scan.nextLine();
        System.out.println("Input Address: ");
        upAddress = scan.nextLine();
        System.out.println("Input Contact Number: ");
        upContact = scan.nextLine();
    }
    public void displayInfo()
    {
        System.out.println("People Information::");
        System.out.println("Name="+upName);
        System.out.println("Address =" +upAddress);
        System.out.println("Contact Number =" +upContact);
    }
}
```



```

class Student extends UniversityPeople
{
private String stDept, stCourse;
private int stSemester,stRollNo;

public void inputStudentInfo(Scanner scan)
    {
System.out.println("Input Information of the Student::");
System.out.println("Input Student Department: ");
stDept = scan.nextLine();

System.out.println("Input Student Course: ");
stCourse = scan.nextLine();
System.out.println("Input Semester: ");

stSemester = scan.nextInt();
System.out.println("Input Student Roll Number: ");

stRollNo = scan.nextInt();
scan.nextLine();

    }

public void displayStudentInfo()
    {
System.out.println("People Type-Student");
System.out.println("Roll Number="+stRollNo);
System.out.println("Department of the Student =" +stDept);
System.out.println("Course of the Student="+stCourse);
System.out.println("Semester of the Student="+stSemester);

    }
}

class Staff extends UniversityPeople
{
private String staDept,staDOJ;
private int staEL,staCL;

public void inputStaffInfo(Scanner scan)
    {

```

```

System.out.println("Input Information of the Staff:");
System.out.println("Input Department of the Staff: ");
staDept = scan.nextLine();

System.out.println("Input Date of Joining of the Staff: ");
staDOJ = scan.nextLine();
System.out.println("Input Available Earn Leave: ");

staEL = scan.nextInt();
System.out.println("Input Available Casual Leave: ");

staCL = scan.nextInt();

    }

public void displayStaffInfo()
{
System.out.println("People Type-Staff");
System.out.println("Department of the Staff =" +staDept);
System.out.println("Date of Joining of the Staff="+staDOJ);
System.out.println("Available Earn Leave the Staff="+staEL);
System.out.println("Available Casual Leave of the Staff="+staCL);

    }
}

class UniversityInfo
{
public static void main(String[] args)
{
Scanner scan = new Scanner(System.in);
Student st1 = new Student();
Staff sta1 = new Staff();
st1.inputInfo(scan);
st1.inputStudentInfo(scan);
st1.displayInfo();
st1.displayStudentInfo();
sta1.inputInfo(scan);
sta1.inputStaffInfo(scan);
sta1.displayInfo();
sta1.displayStaffInfo();
}
}

```

```
}  
}
```

Output of the program:

Input People Information::

Input Name: RakeshRabha

Input Address:Goalpara, Assam

Input Contact Number:5555

Input Information of the Student::

Input Student Department: GUCDOE

Input Student Course: M.A. in English

Input Semester:3

Input Student Roll Number:5

People Information::

Name=RakeshRabha

Address =Goalpara, Assam

Contact Number =5555

People Type-Student

Roll Number=5

Department of the Student =GUCDOE

Course of the Student=M.A. in English

Semester of the Student=3

Input People Information::

Input Name: PrabalGoswami

Input Address:Nalbari, Assam

Input Contact Number:33333

Input Information of the Staff::

Input Department of the Staff: GUCDOE

Input Date of Joining of the Staff: 13-07-2009

Input Available Earn Leave:0

Input Available Casual Leave:7

People Information::

Name=PrabalGoswami

Address =Nalbari, Assam

Contact Number =33333

People Type-Staff

Department of the Staff =GUCDOE

Date of Joining of the Staff=13-07-2009
Available Earn Leave the Staff=0
Available Casual Leave of the Staff=7

In the above program, 'Staff' and 'Student' are two derived classes and both of these classes are derived from the same base class, 'UniversityPeople'. The diagrammatic representation of this Hierarchical Inheritance is presented as follows.

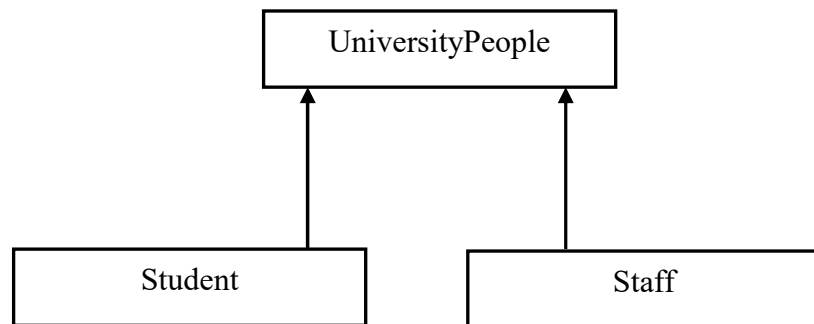


Figure 2.3: Diagrammatic Representation of a Hierarchical Inheritance

2.6.4 Multi-Level Inheritance

When a sub class is derived from another sub class then it is termed as Multi-Level Inheritance. Let us consider the following Java program to understand the concept of Multi-Level Inheritance.

Program 2.5: Java program to demonstrate Multi-Level Inheritance

```
import java.util.Scanner;

class UniversityPeople
{
    private String upName, upAddress, upContact;
    public void inputInfo(Scanner scan)
    {
        System.out.println("Input People Information::");
        System.out.println("Input Name: ");
        upName = scan.nextLine();
        System.out.println("Input Address: ");
        upAddress = scan.nextLine();
    }
}
```

```

System.out.println("Input Contact Number: ");
upContact = scan.nextLine();
    }
    public void displayInfo()
    {
        System.out.println("People Information::");
        System.out.println("Name="+upName);
        System.out.println("Address =" +upAddress);
        System.out.println("Contact Number =" +upContact);
    }
}

class Staff extends UniversityPeople
{
    private String staDept,staDOJ;
    private int staEL,staCL;

    public void inputStaffInfo(Scanner scan)
    {
        System.out.println("Input Information of the Staff::");
        System.out.println("Input Department of the Staff: ");
        staDept = scan.nextLine();

        System.out.println("Input Date of Joining of the Staff: ");
        staDOJ = scan.nextLine();
        System.out.println("Input Available Earn Leave: ");

        staEL = scan.nextInt();
        System.out.println("Input Available Casual Leave: ");

        staCL = scan.nextInt();
        scan.nextLine();

    }

    public void displayStaffInfo()
    {
        System.out.println("People Type-Staff");
        System.out.println("Department of the Staff =" +staDept);
        System.out.println("Date of Joining of the Staff="+staDOJ);
        System.out.println("Available Earn Leave the Staff="+staEL);
    }
}

```

```

        System.out.println("Available Casual Leave of the Staff="+staCL);

    }
}

class TeachingStaff extends Staff
{
    private String tstaDesig;
    private double tstaBasic,tstaDA;

    public void inputTeachingStaffInfo(Scanner scan)
    {
        System.out.println("Input Information of the Teaching Staff:");
        System.out.println("Input Designation of the Teaching Staff: ");
        tstaDesig = scan.nextLine();

        System.out.println("Input Basic Salary of the Teaching Staff: ");
        tstaBasic = scan.nextInt();
        System.out.println("Input Dearness Allowance: ");

        tstaDA = scan.nextInt();

    }

    public void displayTeachingStaffInfo()
    {
        System.out.println("Staff Type-Teaching");
        System.out.println("Designation of the Staff="+tstaDesig);
        System.out.println("Basic Salary="+tstaBasic);
        System.out.println("Dearness Allowance="+tstaDA);

    }
}

class UniversityInfo
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);

        TeachingStaff tsta1 = new TeachingStaff();
    }
}

```

```

tsta1.inputInfo(scan);
tsta1.inputStaffInfo(scan);
tsta1.inputTeachingStaffInfo(scan);

tsta1.displayInfo();
tsta1.displayStaffInfo();
tsta1.displayTeachingStaffInfo();
}
}

```

Output of the program:

Input People Information::

```

Input Name: AnkurDeka
Input Address:Dibrugarh, Assam
Input Contact Number: 44444
Input Information of the Staff::
Input Department of the Staff: GUCDOE
Input Date of Joining of the Staff: 12-11-2009
Input Available Earn Leave: 0

```

Input Available Casual Leave: 6

Input Information of the Teaching Staff::

```

Input Designation of the Teaching Staff: Assistant Professor
Input Basic Salary of the Teaching Staff: 78600
Input Dearness Allowance: 53

```

People Information::

```

Name=AnkurDeka
Address =Dibrugarh, Assam
Contact Number =44444

```

People Type-Staff

```

Department of the Staff =GUCDOE
Date of Joining of the Staff=12-11-2009
Available Earn Leave the Staff=0

```

Available Casual Leave of the Staff=6

Staff Type-Teaching

```

Designation of the Staff =Assistant Professor
Basic Salary=78600
Dearness Allowance=53

```

In the above program it is observed that 'TeachingStaff' is a sub class derived from the class, 'Staff' that is again a sub class derived from the base class, 'UniversityPeople'. It means that 'UniversityPeople' is the grandparent class of 'TeachingStaff'. The diagrammatic representation of this Multi-Level Inheritance is presented as follows.

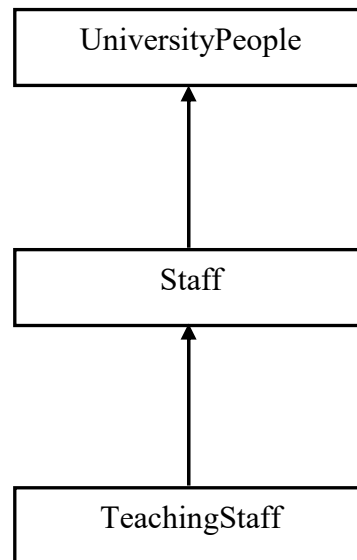


Figure 2.4: Diagrammatic Representation of a Multi-Level Inheritance

2.6.5 Hybrid Inheritance

We have already learnt about four types of Inheritance in the earlier sub-sections. Now, if more than one types of inheritance among these four types of Inheritance are combined to implement an Inheritance then it is termed as Hybrid Inheritance. Let us consider the following Java program to understand the concept of Hybrid Inheritance.

Program 2.6: Java program to demonstrate Hybrid Inheritance

```
import java.util.Scanner;

class UniversityPeople
{
    private String upName, upAddress, upContact;
    public void inputInfo(Scanner scan)
```



```

    {
        System.out.println("Input Information::");
        System.out.println("Input Name: ");
        upName = scan.nextLine();
        System.out.println("Input Address: ");
        upAddress = scan.nextLine();
        System.out.println("Input Contact Number: ");
        upContact = scan.nextLine();
    }
    public void displayInfo()
    {
        System.out.println("People Information::");
        System.out.println("Name="+upName);
        System.out.println("Address =" +upAddress);
        System.out.println("Contact Number =" +upContact);
    }
}
class Student extends UniversityPeople
{
    private String stDept, stCourse;
    private int stSemester,stRollNo;

    public void inputStudentInfo(Scanner scan)
    {
        System.out.println("Input Information of the Student::");
        System.out.println("Input Student Department: ");
        stDept = scan.nextLine();

        System.out.println("Input Student Course: ");
        stCourse = scan.nextLine();
        System.out.println("Input Semester: ");

        stSemester = scan.nextInt();
        System.out.println("Input Student Roll Number: ");

        stRollNo = scan.nextInt();
        scan.nextLine();

    }

    public void displayStudentInfo()

```

```

        {
System.out.println("People Type-Student");
System.out.println("Roll Number="+stRollNo);
System.out.println("Department of the Stduent =" +stDept);
System.out.println("Course of the Student="+stCourse);
System.out.println("Semester of the Student="+stSemester);

        }
    }

class Staff extends UniversityPeople
{
private String staDept,staDOJ;
private int staEL,staCL;

public void inputStaffInfo(Scanner scan)
{
System.out.println("Input Information of the Staff:");
System.out.println("Input Department of the Staff: ");
staDept = scan.nextLine();

System.out.println("Input Date of Joining of the Staff: ");
staDOJ = scan.nextLine();
System.out.println("Input Available Earn Leave: ");

staEL = scan.nextInt();
System.out.println("Input Available Casual Leave: ");

staCL = scan.nextInt();
scan.nextLine();

}

public void displayStaffInfo()
{
System.out.println("People Type-Staff");
System.out.println("Department of the Staff =" +staDept);
System.out.println("Date of Joining of the Staff="+staDOJ);
System.out.println("Available Earn Leave the Staff="+staEL);
System.out.println("Available Casual Leave of the Staff="+staCL);
}
}

```

```

    }
}
class TeachingStaff extends Staff
{
private String tstaDesig;
private double tstaBasic,tstaDA;

public void inputTeachingStaffInfo(Scanner scan)
{
System.out.println("Input Information of the Teaching Staff:");
System.out.println("Input Designation of the Teaching Staff: ");
tstaDesig = scan.nextLine();

System.out.println("Input Basic Salary of the Teaching Staff: ");
tstaBasic = scan.nextInt();
System.out.println("Input Dearness Allowance: ");

tstaDA = scan.nextInt();
scan.nextLine();

}

    public void displayTeachingStaffInfo()
    {
System.out.println("Staff Type-Teaching");
System.out.println("Designation of the Staff="+tstaDesig);
System.out.println("Basic Salary="+tstaBasic);
System.out.println("Dearness Allowance="+tstaDA);

    }
}

class UniversityInfo
{
public static void main(String[] args)
{
Scanner scan = new Scanner(System.in);

TeachingStaff tsta1 = new TeachingStaff();

```

```

tsta1.inputInfo(scan);
tsta1.inputStaffInfo(scan);
tsta1.inputTeachingStaffInfo(scan);

tsta1.displayInfo();
tsta1.displayStaffInfo();
tsta1.displayTeachingStaffInfo();

    Student tst1 = new Student();
tst1.inputInfo(scan);
tst1.inputStudentInfo(scan);

tst1.displayInfo();
tst1.displayStudentInfo();

    }
}

```

Output of the program:

Input People Information::

Input Name: AnkurDeka

Input Address:Dibrugarh, Assam

Input Contact Number: 44444

Input Information of the Staff::

Input Department of the Staff: GUCDOE

Input Date of Joining of the Staff: 12-11-2009

Input Available Earn Leave: 0

Input Available Casual Leave: 6

Input Information of the Teaching Staff::

Input Designation of the Teaching Staff: Assistant Professor

Input Basic Salary of the Teaching Staff: 78600

Input Dearness Allowance: 53

People Information::

Name=AnkurDeka

Address =Dibrugarh, Assam

Contact Number =44444

People Type-Staff

Department of the Staff =GUCDOE

Date of Joining of the Staff=12-11-2009

Available Earn Leave the Staff=0
Available Casual Leave of the Staff=6
Staff Type-Teaching
Designation of the Staff=Assistant Professor
Basic Salary=78600
Dearness Allowance=53

Input People Information::

Input Name:AmitSarma

Input Address: Guwahati, Assam

Input Contact Number: 22222

Input Information of the Student::

Input Student Department: GUCDOE

Input Student Course: M.A. in Assamese

Input Semester: 2

Input Student Roll Number: 25

People Information::

Name=AmitSarma

Address =Guwahati, Assam

Contact Number =22222

People Type-Student

Roll Number=25

Department of the Student =GUCDOE

Course of the Student=M.A. in Assamese

Semester of the Student=2

In the above program, it is observed that 'Student' and 'Staff' are two derived classes derived from the same base class, 'UniversityPeople'. So, a Hierarchical Inheritance is noticed here. Again, it is also observed that 'TeachingStaff' is derived from 'Staff' and 'Staff' is derived from 'UniversityPeople'. Here, a Multi-Level Inheritance is also noticed. This combination of two types of Inheritance is an example of Hybrid Inheritance. The diagrammatic representation of this Hybrid Inheritance is presented as follows.

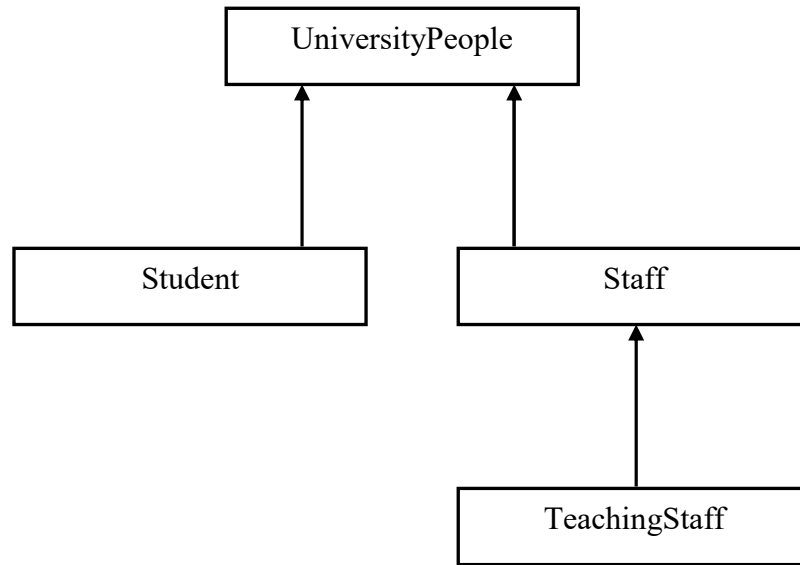


Figure 2.5: Diagrammatic Representation of a Hybrid Inheritance

2.7 VIRTUAL BASE CLASS

Let us consider a situation where a child class is derived from two base classes and both of these base classes are derived from a common base class. As a result, the child class object will receive two instances of its grandparent class through its base classes. In this situation, ambiguity issues may occur if members of the grandparent class are required to be accessed by the grandchild class objects. In C++, the concept of virtual base class is provided to solve this problem. In C++, when multiple derived classes are created from a common base class then in the definitions of the derived classes, the base class is declared as virtual by using the key word '*virtual*'. As a result, if a child class is derived from these derived classes then the child class object will receive a single instance of its grandparent class. As a result, ambiguity and memory duplications can be avoided.

Program 2.7: C++ program to demonstrate the concept of Virtual Base class.

```
#include<iostream.h>
```

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

```
class class1// Grandparent class
```

```
{  
protected:  
    int num;  
public:
```

```
    class1()  
    {  
        num=10;  
    }
```

```
};
```

```
class class2:virtual public class1// Base class is declared as virtual
```

```
{  
public:
```

```
    class2()  
    {  
        num=20;  
    }
```

```
};
```

```
class class3:virtual public class1// Base class is declared as virtual
```

```
{  
public:
```

```
    class3()  
    {  
        num=30;  
    }
```

```
};
```

```
class class4: public class2,public class3// Grandchild class
```

```
{  
public:  
    void display()  
    {
```

```

        cout<<"\n Value in num ="<<num;
    }
};

int main()
{
    class4 c1;
    clrscr();
    c1.display();
    getch();
    return(0);
}

```

Output of the program:

Value in num = 30

In the above program, class1, class2, class3 and class4 are four classes where class2 and class3 are derived from class1 and class4 is derived from class2 and class3. So class4 is the grand child class of class1. In this situation, objects of class4 should receive two instances of class1 through class2 and class3. But in the definitions of class2 and class3, the base class, class1 is declared as virtual by using the keyword, '*virtual*'. As a result, objects of class4 will receive only one instance of class1. But if class1 is not declared as virtual in the definitions of class2 and class3, then ambiguity issue will occur and the following error will occur if we try to compile the program.

- **Error virtualBase.CPP 41: Member is ambiguous: 'class1::num' and 'class1::num'**

CHECK YOUR PROGRESS

1. Choose the correct option
 - (a) ____ members of a base class cannot be inherited to its derived classes.
 - (i) Public

- (ii) Private
 - (iii) Protected
 - (iv) None of the above
- (b) _____ members of a base class can be inherited to its derived classes.
- (i) Public
 - (ii) Protected
 - (iii) Private
 - (iv) Both (i) and (ii)
- (c) In C++, if the Visibility-Mode of an Inheritance is **'protected'** then _____.
- (i) Public members of the base class will be inherited to the derived class as public members.
 - (ii) Public members of the base class will be inherited to the derived class as protected members.
 - (iii) Public members of the base class will be inherited to the derived class as private members.
 - (iv) None of the above
- (d) Which of the following is an advantage of Inheritance?
- (i) Code reusability
 - (ii) Decrease in application development time
 - (iii) New features can be easily added to an application
 - (iv) All of the above
- (e) Which of the following is not an advantage of Inheritance?
- (i) Implementation of real-world relationships
 - (ii) Increase in code redundancy
 - (iii) Extensibility
 - (iv) None of the above
- (f) Upcasting means _____.
- (i) Use a pointer or reference to refer an object of a class.
 - (ii) Use a child class pointer or a child class reference for pointing or referring a base class object.
 - (iii) Use a base class pointer or a base class reference for pointing or referring a derived class object.
 - (iv) None of the above

- (g) Which of the following is not a type of Inheritance?
- (i) Hybrid Inheritance
 - (ii) Multiple Inheritance
 - (iii) Complete Inheritance
 - (iv) Hierarchical Inheritance
- (h) Which of the following statement is true for Hierarchical Inheritance?
- (i) Multiple child classes are derived from the same base class.
 - (ii) One child class is derived from multiple base classes.
 - (iii) A derived class is derived from another derived class.
 - (iv) None of the above
- (i) Which of the following statement is true for Multi-Level Inheritance?
- (i) A derived class is derived from multiple base classes.
 - (ii) A derived class is derived from another derived class.
 - (iii) Multiple derived classes derived from one base class
 - (iv) None of the above
- (j) Which of the following Inheritance is implemented by combining more than one type of Inheritance?
- (i) Multiple Inheritance
 - (ii) Multi-Level Inheritance
 - (iii) Hybrid Inheritance
 - (iv) None of the above
- (k) Which of the following is not supported in Java Programming?
- (i) Multiple Inheritance
 - (ii) Multi-level Inheritance
 - (iii) Single Inheritance

(iv) None of the above

2.8 SUMMING UP

- Inheritance is one of the basic properties of Object Oriented Programming (OOP). Inheritance allows creating new sub or child classes from the already existing classes. In this process, the already existing classes are called as base class and the newly created classes from a base class are called as derived classes.
- Private members of a base class cannot be inherited to its derived classes. Public and protected members of a base class can be inherited to its derived classes. In Java, the members of a base class with 'default' access specifier can be inherited to its derived classes that are available in the same package.
- Advantages of Inheritance are: (a) Reusability of code, (b) Extensibility, (c) Decrease in application development time, (d) Implementation of real-world relationships, (e) Easier code maintenance and modification (f) Implementation of run-time polymorphism (g) Decrease in code redundancy.
- In Inheritance, casting up the hierarchy refers to use a base class pointer or a base class reference for pointing or referring a derived class object.
- Five types of Inheritance supported in OOP are: (a)Single Inheritance, (b)Multiple Inheritance, (c)Hierarchical Inheritance, (d)Multi-Level Inheritance, (e)Hybrid Inheritance.
- In Single Inheritance, one child class is derived from only one base class.

- In Multiple Inheritance, a sub class is derived from more than one base class.
- In Hierarchical Inheritance, more than one sub classes are derived from a single base class.
- In Multi-Level Inheritance, a sub class is derived from another sub class.
- In a Hybrid Inheritance, more than one types of inheritance are combined to implement the Inheritance.
- In C++, when multiple derived classes are created from a common base class and a child class is again derived from these derived classes then in the definitions of these derived classes, the base class is declared as virtual by using the key word '*virtual*'. As a result, objects of the child class derived from these derived classes will receive a single instance of its grandparent class. As a result, ambiguity and memory duplications can be avoided.

ANSWERS TO CHECK YOUR PROGRESS

1.

- (a) (ii) Private
- (b) (iv) Both (i) and (ii)
- (c)(ii) Public members of the base class will be inherited to the derived class as protected members.
- (d) (iv) All of the above
- (e) (ii) Increase in code redundancy
- (f) (iii) Use a base class pointer or a base class reference for pointing or referring a derived class object.
- (g) (iii) Complete Inheritance
- (h) (i) Multiple child classes are derived from the same base class.
- (i) (ii) A derived class is derived from another derived class.
- (j) (iii) Hybrid Inheritance
- (k) (i) Multiple Inheritance

2.9 POSSIBLE QUESTIONS

- 1) Define Inheritance. Explain the importance of Inheritance in Object Oriented Programming (OOP) with suitable examples.
- 2) Write down the relationship between access specifiers and Inheritance.
- 3) What is Upcasting? Give example.
- 4) Explain different types of Inheritance with suitable examples.
- 5) Write a short note on Virtual Base class.

2.10 REFERENCES AND SUGGESTED READINGS

- 1) Venugopal, K. R., Rajkumar, Ravishankar, T. *Mastering C++*. Tata McGraw-Hill Education, 2001.
- 2) Balagurusamy, E. *Object Oriented Programming with C++*. Tata McGraw-Hill, 2006
- 3) Jana, Debasish. *Java and object-oriented programming paradigm*. PHI Learning Pvt. Ltd., 2005.
- 4) Schildt, Herbert. *Java: the complete reference*. McGraw-Hill Education Group, 2014.

---x---

UNIT 3: POLYMORPHISM

Unit Structure:

- 3.1 Introduction
- 3.2 Objectives
- 3.3 Introduction to Polymorphism
- 3.4 Compile Time polymorphism
 - 3.4.1 Function Overloading
 - 3.4.2 Operator Overloading
 - 3.4.3 Static Binding
- 3.5 Run time Polymorphism
 - 3.5.1 Virtual Function
 - 3.5.1.1 Pure Virtual Function
 - 3.5.2 Dynamic Binding
- 3.6 Summing Up
- 3.7 Possible Questions
- 3.8 References and Suggested Readings

3.1 INTRODUCTION

Polymorphism is one of the important properties of Object Oriented Programming (OOP). Origin of the word, 'Polymorphism' is two Greek words, 'Poly' and 'Morphe' where 'Poly' means 'many' and 'Morphe' means 'form'. In this unit, we are going to discuss about Polymorphism in OOP and its different types. Different concepts related to Polymorphism will also be discussed in this unit.

3.2 OBJECTIVES

After reading this unit, you are expected to be able to learn:

- About Polymorphism.

- About Compile time polymorphism and static binding.
- About Function overloading.
- About Operator overloading.
- About Runtime polymorphism and Dynamic binding.
- About Virtual function.
- About Pure virtual function.
- What is Abstract class?

3.3 INTRODUCTION TO POLYMORPHISM

Polymorphism is an important property of Object Oriented Programming (OOP) that allows same function name for multiple functions with different functionalities and defining new operations for an operator. Due to Polymorphism, the code extensibility of an OOP language becomes simpler and better. Polymorphism improves code reusability, readability and maintainability. Dynamic binding is supported by Polymorphism in OOP so that appropriate function can be called at runtime. But it may lead to the performance issues as this process may require additional computations at runtime.

There are two types of Polymorphism available in OOP. These two categories of Polymorphism are Compile-time polymorphism and Run-time polymorphism.

3.4 COMPILE-TIME POLYMORPHISM

We have already learnt that in OOP, Polymorphism permits to use the same function name for multiple functions with different functionalities and to define new operations for an operator. In this process, if there are multiple functions with same function name then in case of compile time polymorphism, which function will be called at the time of function call will be decided by the compiler at compile time depending upon the number of parameters or types of parameters or both available in the function call. Similarly, if a new operation is defined for an operator then the compiler will decide

about the execution of this operation at compile time depending upon the operands of that operator. So, Compile-time polymorphism is divided into two types that are Function overloading and Operator overloading.

3.4.1 Function Overloading

When more than one function are available in a program with same function name but their number of parameters or types of corresponding parameters or both are different from each other then it is referred as Function overloading. Each of these functions will perform different jobs. Function overloading can improve the readability of a program.

Let us consider the following Java program to acquire a clear idea about Function overloading.

Program 3.1: Java program to demonstrate Function overloading

```
import java.util.Scanner;

class Average
{
    private int i, total;

    //Function overloading

    public int averageEst(int N1, int N2, int N3)
    {
        return((N1+N2+N3)/3);
    }

    public float averageEst(float N1, float N2, float N3)
    {
        return((N1+N2+N3)/3);
    }

    public int averageEst(int[] arrNum, int N)
    {
        total =0;
```



```

for(i = 0;i<N;i++)
{
total+=arrNum[i];
}
return( total/N);
}
}

```

```

class calAverage
{
public static void main(String[] args)
{
int num1, num2, num3, avg;
int[] arrNum = new int[100];
int i, N;
float rnum1, rnum2, rnum3, ravg;
Scanner scan = new Scanner(System.in);
System.out.println("Enter three integer numbers: ");
System.out.println("Enter the value for num1= ");

    num1 = scan.nextInt();
scan.nextLine();

System.out.println("Enter the value for num2= ");
    num2 = scan.nextInt();
scan.nextLine();

System.out.println("Enter the value for num3= ");
    num3 = scan.nextInt();
scan.nextLine();

    Average average1=new Average();
avg= average1.averageEst(num1,num2,num3);
System.out.println("Estimated average of three integer numbers is="
+avg);

System.out.println("Enter three real numbers: ");
System.out.println("Enter the value for rnum1= ");
    rnum1 = scan.nextFloat();
scan.nextLine();

```

```

System.out.println("Enter the value for rnum2= ");
    rnum2 = scan.nextFloat();
scan.nextLine();

System.out.println("Enter the value for rnum3= ");
    rnum3 = scan.nextFloat();
avg= average1.averageEst(rnum1,rnum2,rnum3);
System.out.println("Estimated average of three integer numbers is="
+avg);

scan.nextLine();
System.out.println(" Enter the total number of data to be stored in
the array=");
    N = scan.nextInt();
scan.nextLine();

if(N>200)
System.out.println("Wrong input");
else
{
System.out.println("Enter "+N+" number of integer numbers into the
array::");
for(i = 0;i<N;i++)
{
    System.out.println("Enter "+(i+1)+"th data into the array=");
    arrNum[i] = scan.nextInt();
scan.nextLine();

    }

avg=average1.averageEst(arrNum, N);
System.out.println("Average of the numbers stored in the array
is="+avg);

scan.close();

    }
}
}

```

Output of the program:

Enter three integer numbers:
Enter the value for num1=12
Enter the value for num2=23
Enter the value for num3=45
Estimated average of three integer numbers is=26
Enter three real numbers:
Enter the value for rnum1=12.8
Enter the value for rnum2=67.5
Enter the value for rnum3=90.23
Estimated average of three integer numbers is=56.843334
Enter the total number of data to be stored in the array=3
Enter 3 number of integer numbers into the array::
Enter 1th data into the array=10
Enter 2th data into the array=20
Enter 3th data into the array=30
Average of the numbers stored in the array is=20

In **Program 3.1**, three member functions are defined in the class 'Average' with the same function name, 'averageEst' where each of the function is different from the other two functions in terms of either number of arguments or types of the corresponding arguments. From the output of the program, it is observed that, depending upon the arguments, the matching function with function name, 'averageEst' is invoked.

3.4.2 Operator Overloading

Different operators are available in a computer programming language and each of these operators can perform specific operations on only primitive or built-in data types. For example, Addition (+) operator is a binary operator and it is used for arithmetic addition. In Object Oriented Programming (OOP), additional operations can be defined for an operator so that it can also be used on user-defined data types or user-defined classes and this process is referred as Operator overloading. Due to Operator overloading, one operator can be utilized to perform operations on both primitive data types and user defined data types. So, Operator overloading can improve the extensibility of an OOP language.

In an OOP language, existing operators can only be overloaded. It means that no new operators can be created in the process of Operator overloading. The basic operation of an operator cannot be changed by Operator overloading. Syntax rules of the operators are also followed when they are overloaded. All operators available in an OOP language cannot be overloaded. For example, in C++, ::(Scope Resolution Operator) is one of the operators that cannot be overloaded. The compiler can identify the new operation of an overloaded operator at the time of compilation.

All OOP languages do not support Operator overloading. For example, Operator overloading is not supported directly in JAVA. In C++, Operator overloading is implemented by defining special member functions or friend functions in a class. Details of Operator overloading in C++ will be discussed in the later chapter.

3.4.3 Static Binding

Connecting a function call to its corresponding function body or function definition is termed as binding in case of computer programming. If the binding can be performed by the compiler at the time of compilation then it is referred as static binding. In case of static binding, the compiler can be able to identify and collect all the required information at the compile time to link a particular function call to its corresponding function body. As a result, due to static binding the efficiency of a program is improved. But the flexibility of a program may be degraded due to static binding. Compile-time polymorphism is executed by static binding.

3.5 RUN-TIME POLYMORPHISM

In OOP, Run-time Polymorphism is associated with Inheritance. When both a base class and its derived class contain a function with same function prototypes and an object of the derived class is used to call that function then the function in the derived class will be called by overriding the function available in the base class. It is referred as function overriding. Concept of Virtual function can be used for function overriding in OOP. If Virtual function is used for function overriding then linking of the function call with its appropriate function body is determined at the run-time.

So, function overriding by using virtual function is a Run-time Polymorphism.

3.5.1 Virtual Function

In OOP, Virtual function is used to implement Runtime Polymorphism. In C++, functions available in a base class are declared as Virtual function by using '*virtual*' keyword so that at runtime these functions can be overridden by the corresponding functions with same signature that are available as member functions in the derived classes. The functionality of a virtual function is different from the functionality of its corresponding function with same signature that is available in the derived class. In C++, a Virtual function can be overridden when it is called by using a base class reference or a pointer that point to or refer a derived class object. The implementation of function overriding using Virtual function in C++ is discussed in a later chapter.

In Java, '*virtual*' keyword is not available and it is not required to declare a function as Virtual function because all instance functions in Java are by default Virtual functions except static, final, and private functions. In Java, a virtual function available in a base class can be overridden by a function with same signature available in its derived class by calling that function using a reference of the base class which is referring an object of the derived class. Let us consider the following Java program to acquire a clear idea about Function overriding using Virtual function.

Program 3.2: Java program to demonstrate Function overriding

```
import java.util.Scanner;

class Student
{
    private String sName, sAddress, sCourse;
    private int rollNo;
    public void inputInfo()
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter Student Name: ");
```

```

sName = scan.next();
scan.nextLine();

System.out.println("Enter Student Address: ");
sAddress = scan.next();
scan.nextLine();
System.out.println("Enter Student Course: ");

sCourse = scan.next();
scan.nextLine();
System.out.println("Enter Student Roll Number: ");

rollNo = scan.nextInt();
scan.nextLine();
scan.close();
    }

public void displayInfo()
{
System.out.println("Entered Student Information::");
System.out.println("Stduent Name="+sName);
System.out.println("Student Address="+sAddress);
System.out.println("The course of the Student="+sCourse);
System.out.println("Student Roll Number="+rollNo);

    }
}

class Examination extends Student
{
private int markP1, markP2, markP3, markP4;
private int markTotal, per;
public void inputInfo()
{
Scanner scan = new Scanner(System.in);
System.out.println("Enter Marks obtained in the Examination::");
System.out.println("Enter marks obtained in Paper1=");
markP1 = scan.nextInt();
scan.nextLine();

System.out.println("Enter marks obtained in Paper2=");

```

```

markP2 = scan.nextInt();
scan.nextLine();

System.out.println("Enter marks obtained in Paper3=");
markP3 = scan.nextInt();
scan.nextLine();

System.out.println("Enter marks obtained in Paper4=");
markP4 = scan.nextInt();
scan.nextLine();

markTotal = markP1+markP2+markP3+markP4;
per = markTotal/4;
scan.close();
    }

public void displayInfo()
    {
System.out.println("Student's Examination Information::");
System.out.println("Marks obtained in Paper1="+markP1);
System.out.println("Marks obtained in Paper2="+markP2);
System.out.println("Marks obtained in Paper3="+markP3);
System.out.println("Marks obtained in Paper4="+markP4);
System.out.println("Total Marks obtained="+markTotal);
System.out.println("Obtained Percentage="+per);

    }
}

class StudentInfo
{
public static void main(String[] args)
{
Student st = new Examination();
st.inputInfo();
st.displayInfo();
}
}

```

Output of the program:

Enter Marks obtained in the Examination::

```
Enter marks obtained in Paper1=78
Enter marks obtained in Paper2=90
Enter marks obtained in Paper3=80
Enter marks obtained in Paper4=67
Student's Examination Information::
Marks obtained in Paper1=78
Marks obtained in Paper2=90
Marks obtained in Paper3=80
Marks obtained in Paper4=67
Total Marks obtained=315
Obtained Percentage=78
```

In **program 3.2**, the class, 'Examination' is extended from the base class, 'Student'. The derived class, 'Examination' contain two member functions with same function prototypes to the two member functions available in the base class, 'Student'. These two functions are 'inputInfo()' and 'displayInfo()'. From the output of the program, it is observed that when a reference of the base class, 'Student' is used to refer an object of the derived class, 'Examination' then the member functions ('inputInfo()' and 'displayInfo()') of the base class are overridden by the member functions('inputInfo()' and 'displayInfo()') of the derived class.

3.5.1.1 Pure Virtual Function

In C++, Pure virtual function is a virtual function that is declared in a base class but its functionality is not defined in that base class. The functionality of a Pure virtual function must be defined in the derived class of that base class or it has to be declared again as a Pure virtual function in that derived class. In Java, the concept of Pure virtual function is achieved by Abstract method. In Java, an Abstract method is also a method which is declared in a class but its functionality is not defined in that class. The functionality of an Abstract method must be defined in the derived class of the class where the Abstract method is declared or it must be again declared as Abstract in that derived class. '*abstract*' keyword is used to declare a method as Abstract method.

In C++, if a class contains one or more Pure virtual functions then that class is referred as Abstract class. In Java, A base class can

be declared as Abstract class by using the keyword, '*abstract*'. Creation of object of an Abstract class is not possible. In Java, if a class contains any abstract method then that class must be declared as abstract class. In OOP, an Abstract class is considered as a blueprint that can be used to derive different classes with different new features and functionalities.

The syntax of writing Pure virtual function in C++ is presented as follows.

```
class Class_Name
{
public:
    virtual Return_Type Function_Name(Argument List) = 0;

};
```

Let us consider the following C++ program to acquire a clear idea about the implementation of Pure virtual function.

Program 3.3:C++ program to demonstrate Pure virtual function

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

class calArea
{
public:
    virtual void readInfo( ) = 0;        // Pure virtual function
    virtual void displayArea( ) = 0;    // Pure virtual function
};

class Circle : public calArea
{
private:
    float radius;
public:
    void readInfo();
    void displayArea();
};
```

```
};
```

```
void Circle :: readInfo()
{
    cout<< "\n ****Area calculation for Circle****";
    cout<< "\n Enter the radius of the Circle =";
    cin>> radius;
    cout<< "\n";

}
```

```
void Circle :: displayArea()
{
    cout<< "\n Radius of the Circle=";
    cout<< radius;
    cout<< "\n Area of the Circle=";
    cout<< 3.1415*radius*radius; // Value of p=3.1415
    cout<< "\n";
}
```

```
class Square : public calArea
{
private:
    floats Length;
public:
    void readInfo();
    void displayArea();
};
```

```
void Square :: readInfo()
{
    cout<< "\n ****Area calculation for Square****";
    cout<< "\n Enter the length of one side in the Square =";
    cin>>sLength;
    cout<< "\n";

}
```

```
void Square :: displayArea()
{
```

```

cout<< "\n Length of one side in the square =";
cout<<sLength;
cout<< "\n Area of the square =";
cout<<sLength*sLength;
cout<< "\n";
}

class Trapezoid : public calArea
{
private:
float base1, base2, vHeight;
public:
void readInfo();
void displayArea();
};

void Trapezoid :: readInfo()
{
cout<< "\n****Area calculation for Trapezoid****";
cout<< "\n Enter the First base of the Trapezoid=";
cin>> base1;
cout<< "\n Enter the Second base of the Trapezoid=";
cin>> base2;
cout<< "\n Enter the vertical height of the Trapezoid=";
cin>>vHeight;
cout<< "\n";

}

void Trapezoid :: displayArea( )
{
cout<< "\n First base of the Trapezoid=";
cout<< base1;
cout<< "\n Second base of the Trapezoid=";
cout<< base2;
cout<< "\n Vertical height of the Trapezoid=";
cout<<vHeight;
cout<< "\n Area of the Trapezoid=";
cout<< 0.5*(base1+base2)*vHeight;
cout<< "\n";
}

```

```

int main( )
{
    calArea *cA;
    Circle Cr;
    Square Sr;
    Trapezoid Tr;

    clrscr( );
    cA = &Cr;
    cA->readInfo();
    cA->displayArea();

    cA = &Sr;
    cA->readInfo();
    cA->displayArea();

    cA = &Tr;
    cA->readInfo();
    cA->displayArea();

    getch();
    return(0);
}

```

Output of the program:

****Area calculation for Circle****

Enter the radius of the Circle =8

Radius of the Circle=8

Area of the Circle=201.056

****Area calculation for Square****

Enter the length of one side in the Square =2

Length of one side in the square =2

Area of the square =4

****Area calculation for Trapezoid****

Enter the First base of the Trapezoid=4

Enter the Second base of the Trapezoid=5

Enter the vertical height of the Trapezoid=6

First base of the Trapezoid=4

Second base of the Trapezoid=5

Vertical height of the Trapezoid=6

Area of the Trapezoid=27

In **program 3.3**, 'readInfo()' and 'displayArea()' are two pure virtual functions declared in the base class, 'calArea'. The definitions of these two pure virtual functions are provided in the derived classes('Circle', 'Square', and 'Trapezoid') of the base class, 'calArea'.

Let us consider the following Java program to acquire a clear idea about the implementation of Abstract method.

Program 3.4:Java program to demonstrate the implementation of Abstract method.

```
import java.util.Scanner;

abstract class calArea      // Abstract class
{

    abstract void readInfo();      // Abstract method
    abstract void displayArea();  // Abstract method
}

class Circle extends calArea
{
    private float radius;
    public void readInfo()
    {
        Scanner scan = new Scanner(System.in);
        System.out.println(" *****Area calculation for Circle*****");
        System.out.println("Enter the radius of the Circle = ");
        radius = scan.nextFloat();
        scan.nextLine();
        scan.close();
    }
}
```

```

public void displayArea()
{
    System.out.println("Radius of the Circle="+radius);
    System.out.println("Area of the Circle="+3.1415*radius*radius);

}
}

class Square extends calArea
{
    private float sLength;
    public void readInfo()
    {
        Scanner scan = new Scanner(System.in);

        System.out.println("**** Area calculation for Square****");
        System.out.println("Enter the length of one side in the Square =");
        sLength=scan.nextFloat();
        scan.nextLine();
        scan.close();

    }
    public void displayArea()
    {
        System.out.println(" Length of one side in the square =" +sLength);
        System.out.println(" Area of the square =" +sLength*sLength );

    }
}

class Trapezoid extends calArea
{
    private float base1, base2, vHeight;

    public void readInfo()
    {

        Scanner scan = new Scanner(System.in);

```

```

System.out.println("**** Area calculation for Trapezoid****");
System.out.println(" Enter the First base of the Trapezoid=");
base1=scan.nextFloat();
scan.nextLine();
System.out.println(" Enter the Second base of the Trapezoid=");
base2=scan.nextFloat();
scan.nextLine();
System.out.println("Enter the vertical height of the Trapezoid=");
vHeight=scan.nextFloat();
scan.nextLine();

scan.close();

}

public void displayArea()
{
System.out.println("First base of the Trapezoid="+base1);
System.out.println("Second base of the Trapezoid="+base2);
System.out.println("Vertical height of the Trapezoid="+vHeight);
System.out.println("Area          of          the
                    Trapezoid="+0.5*(base1+base2)*vHeight);

}
}

class Area
{
public static void main(String[] args)
{
calArea aR=new Trapezoid();
aR.readInfo();
aR.displayArea();
}
}

```

Output of the program:

```

****Area calculation for Trapezoid****
Enter the First base of the Trapezoid=2
Enter the Second base of the Trapezoid=3

```

Enter the vertical height of the Trapezoid=6

First base of the Trapezoid=2.0

Second base of the Trapezoid=3.0

Vertical height of the Trapezoid=6.0

Area of the Trapezoid=15.0

In **program 3.4**, 'readInfo()' and 'displayArea()' are two abstract methods declared in the base class, 'calArea'. The definitions of these two abstract methods are provided in the derived classes('Circle', 'Square', and 'Trapezoid') of the base class, 'calArea'.

3.5.2 Dynamic Binding

Dynamic binding refers the linking of a function call to its appropriate function body at run-time. It is also termed as Late binding. It is observed that in case of function overriding using virtual function, all required information to call the appropriate member function of the derived class is recognized at run time only and a member function of the base class is overridden by the appropriate member function available in the derived class. So, Dynamic binding enables function overriding using virtual function in OOP.

CHECK YOUR PROGRESS

1. Choose the Correct Option

- (a) Find out the false statement regarding Polymorphism.
 - (i) Same name for multiple methods.
 - (ii) Defining new operation for an existing operator.
 - (iii) Same name for multiple variables.
 - (iv) None of the above.
- (b) Which of the following is a Compile-time Polymorphism?
 - (i) Function overloading
 - (ii) Function overriding using virtual function
 - (iii) Operator overriding
 - (iv) None of the above

- (c) Which of the following is a run-time Polymorphism?
- (i) Function overloading
 - (ii) Function overriding using virtual function
 - (iii) Operator overriding
 - (iv) None of the above
- (d) Function overriding can be implemented by using ____.
- (i) inline function
 - (ii) constructor
 - (iii) virtual function
 - (iv) None of the above
- (e) Defining additional operation for an existing operator is termed as ____.
- (i) Function overloading
 - (ii) Operator overloading
 - (iii) Operator overriding
 - (iv) None of the above
- (f) If connecting a function call to its corresponding function body is performed by the compiler at the time of compilation then it is referred as ____.
- (i) Static binding
 - (ii) Dynamic binding
 - (iii) Virtual binding
 - (iv) None of the above
- (g) In C++, ____ is a keyword used to declare a base class member function as virtual?
- (i) abstract
 - (ii) vir
 - (iii) virtual
 - (iv) None of the above
- (h) Which of the following types of instance function is not virtual function in Java?
- (i) static function
 - (ii) final function
 - (iii) private function

- (iv) All of the above
- (i) _____ is a virtual function that does not have any definition.
 - (i) Pure virtual function
 - (ii) Abstract virtual function
 - (iii) Inline virtual function
 - (iv) None of the above
- (j) An _____ class contains one or more Pure virtual functions.
 - (i) Static
 - (ii) Private
 - (iii) Abstract
 - (iv) None of the above
- (k) If connecting a function call to its corresponding function body is performed at run-time then it is referred as _____.
 - (i) Static binding
 - (ii) Dynamic binding
 - (iii) Virtual binding
 - (iv) None of the above

5.8 SUMMING UP

Polymorphism is one of the important properties of Object Oriented Programming (OOP) that permits same function name for multiple functions with different functionalities and allows defining new operations for an operator.

Advantages of Polymorphism are: (a) the code extensibility of an OOP language becomes simpler and better (b) code reusability, readability and maintainability improves due to Polymorphism.

Two types of Polymorphism are Compile-time polymorphism and Run-time polymorphism.

In case of compile time polymorphism, which function body will be called at the time of function call will be decided by the compiler at compile time depending upon the number of parameters or types of parameters or both.

When more than one function are available in a program with same function name but their number of parameters or types of corresponding parameters or both are different from each other then it is referred as Function overloading.

In Object Oriented Programming (OOP), additional operations can be defined for an operator so that it can also be used on user-defined data types or user-defined classes and it is termed as Operator overloading. If a new operation is defined for an operator then the compiler will decide about the execution of this operation at compile time depending upon the operands of that operator.

Only existing operators in an OOP language can be overloaded. The basic operation of an operator cannot be changed by Operator overloading and the syntax rules of the operators are also followed when they are overloaded.

All OOP languages do not support Operator overloading. For example, Operator overloading is not supported directly in JAVA.

If linking a function call to its corresponding function body or function definition is performed by the compiler at the time of compilation then it is termed as static binding. Compile-time polymorphism is executed by static binding.

Run-time Polymorphism is associated with Inheritance. When both a base class and its derived class contain a function with same function prototypes and an object of the derived class is used to call that function then the function in the derived class will be called by overriding the function available in the base class. It is termed as function overriding.

The concept of Virtual function is used to implement Runtime Polymorphism in OOP. In C++, functions available in a base class are declared as Virtual function by using '*virtual*' keyword so that at runtime these functions can be overridden by the corresponding function with same function prototype that is available as member function in a derived class. In C++, a Virtual function can be overridden when it is called by using a base class reference or a pointer that point to or refer a derived class object.

In Java, '*virtual*' keyword is not available and it is not required to declare a function as Virtual function because all

instance functions in Java are by default Virtual functions except static, final, and private functions.

In C++, if a virtual function is declared in a base class but its functionality is not defined then it is termed as Pure virtual function. The functionality of a Pure virtual function must be defined in the derived class of that base class or it has to be declared again as a Pure virtual function in that derived class.

In Java, the concept of Pure virtual function is realized by Abstract method. In Java, an Abstract method is also a method which is declared in a class but its functionality is not defined in that class. The functionality of an Abstract method must be defined in the derived class of the class where the Abstract method is declared or it must be again declared as Abstract in that derived class. In Java, '*abstract*' keyword is used to declare a method as Abstract method.

In C++, if a class contains one or more Pure virtual functions then that class is termed as Abstract class. In Java, a base class can be declared as Abstract class by using the keyword, '*abstract*'. Instantiation of object of an Abstract class is not possible. In Java, if a class holds any abstract method then that class must be declared as abstract class.

If the linking of a function call to its appropriate function body is performed at run-time then it is termed as Dynamic binding. In case of function overriding using virtual function, all required information to call the appropriate member function of the derived class is recognized at run time only. So, Dynamic binding enables function overriding using virtual function in OOP.

ANSWERS TO CHECK YOUR PROGRESS

1.

- (a) (iii) Same name for multiple variables.
- (b) (i) Function overloading
- (c) (ii) Function overriding using virtual function
- (d) (iii) virtual function
- (e) (ii) Operator overloading
- (f) (i) Static binding
- (g) (iii) virtual

- (h) (iv) All of the above
- (i) (i) Pure virtual function
- (j) (iii) Abstract
- (k) (ii) Dynamic binding

5.9 POSSIBLE QUESTIONS

- 1) Define Polymorphism. How Polymorphism is useful in OOP? Write down the different types of Polymorphism.
- 2) Define Function overloading. Give example.
- 3) Explain Operator overloading.
- 4) Explain virtual function?
- 5) Explain Function overriding.
- 6) What is Pure virtual function? Give example.
- 7) Explain Abstract class. Give example.

5.10 REFERENCES AND SUGGESTED READINGS

- 1) Venugopal, K. R., Rajkumar, Ravishankar, T. *Mastering C++*. Tata McGraw-Hill Education, 2001.
- 2) Balagurusamy, E. *Object Oriented Programming with C++*. Tata McGraw-Hill, 2006
- 3) Jana, Debasish. *Java and object-oriented programming paradigm*. PHI Learning Pvt. Ltd., 2005.
- 4) Schildt, Herbert. *Java: the complete reference*. McGraw-Hill Education Group, 2014.

---x---

UNIT 4: EXCEPTION HANDLING

Unit Structure:

- 4.1 Introduction
- 4.2 Objectives
- 4.3 Introduction to Exception Handling
- 4.4 Exception Handling Model
 - 4.4.1 Try-Block
 - 4.4.2 Throw-Block
 - 4.4.3 Catch-Block
- 4.5 Exception Handling in Constructor
- 4.6 Exception Handling in Destructor
- 4.7 Handling Uncaught Exceptions
- 4.8 Exceptions in Class Templates
- 4.9 Summing Up
- 4.10 Answers to Check Your Progress
- 4.11 Possible Questions
- 4.12 References and Suggested Readings

4.1 INTRODUCTION

A reliable software application must provide an efficient error handling and fault tolerant mechanism so that it can handle any anomalies and unexpected situations occurred during program execution like occurrence of error due to inappropriate input data, occurrence of any anomaly due to the use of any technique that is not appropriate to handle a new set of input data, occurrence of anomalies due to any defect occurred in the associated hardware component etc. In programming languages like C++ and Java, concept of Exception Handling is provided for implementing efficient error handling and fault tolerant mechanism in a software application. In this unit, we are going to discuss about Exceptions and Exception Handling in Object Oriented Programming (OOP).

4.2 OBJECTIVES

After reading this unit, you are expected to be able to learn:

- What is Exception and Exception Handling in Object Oriented Programming (OOP)?
- About Exception Handling Model.
- About Exception Handling in constructor and destructor.
- How to handle Uncaught Exceptions?
- About Exceptions in class template.

4.3 INTRODUCTION TO EXCEPTION HANDLING

An **exception** can be defined as an anomaly or an unexpected condition or an error that can be occurred at the run-time of a program and it can disrupt the normal execution flow of that program. In general, exceptions can be categorized into two types that are synchronous exception and asynchronous exceptions.

Synchronous exceptions are those exceptions that are occurred due to any inappropriate input data or due to the use of any method in the program that is not appropriate for a new set of input data. For example, runtime errors occurred due to the overflow and underflow conditions, dividing any number by zero etc.

On the other hand, **Asynchronous exceptions** are not occurred due to any part of the corresponding program. These exceptions are occurred due to some external events that are not controlled by the program. For example, occurrence of errors due to any defect occurred in the corresponding hardware components.

In programming languages like C++, Java etc., a facility is provided to handle Synchronous exceptions and this facility is termed as **Exception handling**. Exception handling can be used to detect exceptions and handle them before their occurrence so that they cannot disrupt the normal execution flow of the corresponding program. The part of a program that is written to handle probable exceptions is termed as **Exception handler**. When an exception is occurred during the program execution then the corresponding

Exception handler is called to handle that exception. The job of an Exception handler is to catch the exceptions occurred at the time of program execution and then provides possible solutions to the exceptions or if required, terminates the program execution in a graceful manner.

The advantages of Exception handling are presented in the following points.

- Exception handling mechanism helps to avert system failures and crashes. So, Exception handling mechanism can play an important role to develop software that is efficient in case of fault tolerance and avoidance.
- Exception handling mechanism separates the code to handle errors from the normal code in a program. As a result, readability of the program will be improved and maintainability of the program will be easier.
- Exception handling mechanism can be used to provide appropriate error messages to detect errors occurred in a system.
- Exception handling mechanism can also be used to confirm appropriate resource management in a system.
- Another advantage of Exception handling mechanism is that group of related errors may be handled by applying a single Exception handler.

4.4 EXCEPTION HANDLING MODEL

In case of a programming language, Exception handling is implemented using the Exception handling model. In C++, the Exception handling model is composed of three exception handling constructs. These three constructs are try, throw and catch.

Program 4.1: C++ program to demonstrate the Exception Handling mechanism.

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



```

int main()
{
    int data[100];
    inti =0,N_data;

    cout<<"\n Enter the number of values to be entered=\n";
    cin>>N_data;
    try{// try block
    if(N_data>100)
        throw N_data;// throw exception

    cout<<"\n Enter data into the array::";
        while(i<N_data)
        {
            cout<<"\n Enter "<<i+1<<"th data=";
            cin>>data[i];
            i++;
        }
        cout<<"\n The data available in the array are:";
        for(i =0;i<N_data;i++)
        {
            cout<<"\t"<<data[i];
        }

    }
    catch(int excep1)// catch block
    {
        cout<<"\n Exception: Input value is larger than the Array Size";
    }

    getch();
    return(0);
}

```

Outputs of the program:

Output1:

```

Enter the number of values to be entered= 5
Enter 1th data=45
Enter 2th data=7
Enter 3th data=12
Enter 4th data=11

```

Enter 5th data= 22

The data available in the array are:45 7 12 11 22

Output2:

Enter the number of values to be entered=120

Exception: Input value is larger than the Array Size

Two outputs of the program are presented above. From output1, it is observed that there is no exception raised in the try block and the program is executed in its normal program flow. But from output2, it is observed that an exception is thrown from the try block as the input value as the number of values to be entered into the array is larger than the corresponding array size.

4.4.1 Try

A try block in a program is a group of programming statements that may raise run-time errors or exceptions. The try-block is used to handle exceptions or run-time errors. When an exception is raised in a try block then the program flow will be changed by transferring the program control to the appropriate exception handler associated with the try block. If there is no handler available then the corresponding program will call the function 'terminate ()'. If no exception is raised in the try block then the normal program flow will not be interrupted. In C++, the group of statements in a try block is enclosed by braces ({}) and the try block is started with the keyword, 'try'. The syntax of the try block in C++ is presented as follows.

```
try
{
    //Group of statements that may raise exceptions
}
```

As an example, Program 4.1 can be considered to observe the use of try block.

4.4.2 Throw

The throw construct is used inside functions and try blocks to throw exceptions if runtime-errors are occurred during execution

of the programming statements available in a function or a try block. So, the throw construct is used to indicate the occurrence of a run-time error or any unexpected condition during the execution of a block of programming statements. The basic syntax to use throw construct in C++ is presented as follows.

throw exceptionObj ;

In the above syntax, ‘throw’ is a C++ keyword and ‘exceptionObj’ is an object or a built-in type expression.

In Program 4.1, throw construct is used to throw an exception with an integer argument.

In C++, an exception handler can rethrow a caught exception by invoking the throw construct without any arguments. If an exception is rethrown by the corresponding catch block then the same catch block or any other catch block available in the same group will not catch that rethrown exception. Only the matching catch block available outside the try-catch group can catch the rethrown exception. The syntax for rethrow an exception is presented as follows.

```
try
{
    // Statements in the try block
    throw      exceptionObj ;// Throws an exception
}
catch (TypeExceptionexceptionObj)
{
    // Statements in the catch block
    throw; // Rethrow the exception
}
```

In the above syntax, ‘TypeException’ is the type of the exception class.

In C++, the throw construct can be used to specify a list of exceptions that may be raised directly or indirectly by a function. The syntax for this use of throw construct is presented as follows.

```
returnType functionName([Argument list])
throw(TypeExceptionobj1, TypeExceptionobj2.... )
{
```

```

    // Function body
}

```

In C++, the throw construct can also be used to specify that no exceptions will be occurred in a function. If any exception occurs in that function then the library function ‘abort ()’ is invoked to issue an error message and terminate the corresponding program. The syntax to use the throw construct to specify that no exceptions can be occurred in a function is presented as follows.

```

returnType functionName([Argument list]) throw()
{
    // Function body
}

```

4.4.3 Catch

The catch block in the exception handling mechanism is responsible for providing the exception handler for matching exceptions. A catch block catches matching exceptions and handles the exceptions with the appropriate exception handler. When an exception is thrown inside a try block then immediately the program control is transferred to the appropriate catch block to handle that exception. The catch block is written immediately after the try block. Multiple catch blocks can be provided in a program where each block will be executed only for the matching exceptions. A catch block is enclosed by braces ({ }) and the catch block is started with the keyword, ‘catch’. The syntax of the catch block in C++ is presented as follows.

```

catch (TypeException exceptionObj)
{
    // Body of the exception handler
}

```

In the above syntax, ‘TypeException’ is the type of the exception class object or a built-in type.

As an example, Program 4.1 can be considered to observe the use of catch block.

In C++, one single catch block can be used to handle all the exceptions that may be raised in the try block. The syntax of writing such a catch block is presented as follows.

```
catch(...)
{
    // Body of the exception handler
}
```

In the above syntax, catch block with three dots(...) can handle all the exceptions that may be raised in the try block.

4.5 EXCEPTION HANDLING IN CONSTRUCTOR

Exceptions may also be thrown by a constructor of a class when any unexpected condition or run-time error occurred. When an exception is raised in a constructor then the corresponding object may not be created fully and in this situation, the destructor is invoked to release the partially allocated resources that are allocated during the process of object creation. This process of invoking the destructor is termed as stack unwinding. In C++, if an exception is raised and the program flow is changed then destructors are called for all the objects that are created from the starting point of the try-block.

In inheritance, when an exception is raised in a base class constructor then constructors of the corresponding derived classes will not be invoked. If exceptions are raised in the constructors of both the base class and its derived class then the exception handler of the derived class will be executed before the exception handler of the base class.

Exception handling in constructors provides proper release of resource allocations if any unexpected condition or run-time error occurs during the execution of a constructor. If a constructor throws an exception the exception handling mechanism also ensures that the corresponding program remains in a valid state after the occurrence of the exception.

Program 4.2 C++ program to demonstrate exception handling in constructor.

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

class Search_Array
{
    private:
        int data[100], i, N_data, S_data, flag;
    public:
        Search_Array(int N,int S);//Constructor declaration
        void readArray();
        void searchData();
};

Search_Array::Search_Array(intN,int S)
{
    if(N>100)
        throw N;// Throw exception from the constructor
    N_data=N;
    S_data=S;
    flag=0;

}
void Search_Array::readArray()
{
    i =0;
    cout<<"\n Enter data into the array: ";
    while(i<N_data)
    {
        cout<<"\n Enter "<<i+1<<"th data=";
        cin>>data[i];
        i++;
    }
}
void Search_Array::searchData()
{
    for(i =0;i<N_data;i++)
    {
        if(S_data == data[i])
```

```

        {
        cout<<"\n"<<S_data<<" is avaialble at "<<i+1<<"th position";
        flag=1;
        }
    }
    if(flag==0)
    cout<<"\n"<<S_data<<" is not available in the array";
}

int main()
{

    try{
    Search_ArraysA1(120,45);
    sA1.readArray();
    sA1.searchData();
    }

    catch(int excep1)
    {
    cout<<"\n Exception: Input value is larger than the Array Size";
    }

    getch();
    return(0);
}

```

Output of the program:

Exception: Input value is larger than the Array Size

In the above program, the constructor is called with two integer values that are 120 and 45 where 120 is the number of values to be entered into the array and 45 is the data that will be searched in the array. So, from the output, it is observed that an exception is raised from the constructor as 120 is greater than the size of the array which is the data member of the corresponding object.

4.6 EXCEPTION HANDLING IN DESTRUCTOR

In C++, throwing exceptions from destructors should be performed in special cases only and exception handling should be carefully performed. In general, raising exceptions from destructors should be avoided as it may lead to some undesirable situations. For example, throwing exceptions from destructors may cause memory leaks if the deallocation of the corresponding object is not performed in the correct way. In C++, if an exception is thrown from a destructor during run-time stack unwinding when another exception is already being handled then C++ will terminate the corresponding program by calling 'std::terminate()'.

If it is necessary to throw exception from a destructor then it must be ensured that the exceptions will be caught and handled inside the destructor itself.

4.7 HANDLING UNCAUGHT EXCEPTIONS

If an exception is thrown in a program but there is no matching catch block or exception handler available for handling that exception then this exception will be uncaught. In this situation, necessary actions have to be performed on the occurrence of such uncaught exceptions. In C++, four built-in functions are used to handle the uncaught exceptions. These four functions are (a) 'unexpected()', (b) 'set_unexpected()', (c) 'terminate ()' and (d) 'set_terminate()'.

- **'unexpected()'**: When an exception is raised in a program but it is not listed in the corresponding exception specification then the function 'unexpected()' is called. By default, 'unexpected()' calls the function 'terminate ()'.
- **'set_unexpected()'**: The function 'set_unexpected()' can be used to include a function that will be invoked when an exception is occurred but it is not listed in the corresponding exception specification and this function performs the necessary actions to handle that exception.
- **'terminate ()'**: The function 'terminate()' performs the necessary actions in the process of terminating a program

on the occurrence of uncaught exceptions in that program. By default, 'terminate ()' calls the function 'abort ()' and 'abort ()' will immediately terminate the corresponding program execution.

- **'set_terminate()'**:The function 'set_terminate()' can be used to include a function that will be invoked when uncaught exceptions are raised in a program. This function performs the necessary actions in the process of terminating the corresponding program due to the occurrence of uncaught exceptions.

Program 4.3: C++ program to demonstrate the use of 'set_terminate()' to handle uncaught exception.

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

class Search_Array
{
private:
int data[100], S_data;
int i, N_data, flag;
public:
Search_Array(int N,int S);
void readArray();
void searchData();
};

Search_Array::Search_Array(intN,int S)
{
if(N>100)
throw N;
N_data=N;
S_data=S;
flag=0;

}

void Search_Array::readArray()
{
i=0;
```

```

cout<<"\n Enter data into the array::";
while(i<N_data)
{
cout<<"\n Enter "<<i+1<<"th data=";
cin>>data[i];
i++;
}
}
void Search_Array::searchData()
{
for(i=0;i<N_data;i++)
{
if(S_data == data[i])
{
cout<<"\n"<<S_data<<" is avaialble at "<<i+1<<"th position";
flag=1;
}
}
if(flag==0)
cout<<"\n"<<S_data<<" isnot available in the array";
}

void UcExceptionHandler() // Exception handler for uncaught exception
{
cout<<"\n Exception: Input value is larger than the Array Size";
abort();
}

int main()
{

std::set_terminate(UcExceptionHandler); // For uncaught exception
try{
Search_ArraysA1(102,45);
sA1.readArray();
sA1.searchData();
}
catch(char excep1)
{
cout<<"\n Exception: 'char' type data is Invalid";
}
}

```

```

        getch();
        return(0);
    }

```

Output of the program:

Exception: Input value is larger than the Array Size

This application has requested the Runtime to terminate it in an unusual way.

Please contact the application's support team for more information.

In **Program 4.3**, an exception is thrown from the constructor with an integer argument. But no catch block is available for exceptions thrown with integer argument in the program. So, if any exception is thrown from the constructor then it will be uncaught. In that situation, an exception handler ('UcExceptionHandler()') is defined in the program and 'set_terminate()' is used to invoke it if any uncaught exception is occurred. From the output of the program, it is observed that an uncaught exception is raised and as a result 'UcExceptionHandler()' is invoked.

Program 4.4: C++ program to demonstrate the use of 'set_unexpected()' to handle the exception that is not listed in the exception specification.

```

#include<iostream.h>
#include<conio.h>

class Search_Array
{
private:
    int data[100], S_data , i, N_data, flag;

public:
    Search_Array(intN,int S)throw(); // Constructor that will not
    throw any exception
    void readArray();
    void searchData();

```

```

};

Search_Array::Search_Array(int N,int S) throw()
{
    if(N>100)
        throw N; // Throws exception that is not specified
        N_data=N;
        S_data=S;
        flag=0;

    }
    void Search_Array::readArray()
    {
        i=0;
        cout<<"\n Enter data into the array:.";
        while(i<N_data)
        {
            cout<<"\n Enter "<<i+1<<"th data=";
            cin>>data[i];
            i++;
        }
    }
    void Search_Array::searchData()
    {
        for(i=0;i<N_data;i++)
        {
            if(S_data == data[i])
            {
                cout<<"\n"<<S_data<<" is available at "<<i+1<<"th position";
                flag=1;
            }
        }
        if(flag==0)
            cout<<"\n"<<S_data<<" isnot available in the array";
    }

    void UexExceptionHandler() // Exception handler for unspecified exception
    {
        cout<<"\n Exception: Input value is larger than the Array Size";
    }

```

```

    }

    int main()
    {

std::set_unexpected(UexExceptionHandler); //For unspecified exception
        try{
            Search_ArraysA1(102,45);

            sA1.readArray();
            sA1.searchData();
        }
        catch(int excep1)
        {
            cout<<"\n Exception: Wrong Input";
        }

        getch();
        return(0);
    }

```

Output of the program:

Exception: Input value is larger than the Array Size

This application has requested the Runtime to terminate it in an unusual way.

Please contact the application's support team for more information.

In **Program 4.4**, the constructor of the class, 'Search_Array' is specified with 'throw()' that no exception will be thrown from it. But it is observed that it throws exception when the value of N is greater than 100. Such unspecified exception is handled in this program by the user-defined exception handler, 'UexExceptionHandler()' and 'set_unexpected()' is used to install it in the program. From the output of the program, it is observed that an unspecified exception is raised from the constructor and as a result 'UexExceptionHandler()' is invoked.

4.8 EXCEPTIONS IN CLASS TEMPLATES

In C++, exception handling in class template is performed in the same way that is used in the normal exception handling. Let us consider the following C++ program to understand clearly about the exception handling in class template.

Program 4.5: C++ program to demonstrate the Exception handling in class template.

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

template <class T> // Class template
class Search_Array
{
private:
    T data[100];
    int i, N_data, flag;
public:
    Search_Array(int N)
    {
        if(N>100)
            throw N; // Throw exception
        N_data=N;
        flag = 0;
    }
    void readArray();
    void searchData(T S_data);
};

template<class T>
void Search_Array<T>::readArray()
{
    i=0;
    cout<<"\n Enter data into the array:.";
    while(i<N_data)
    {
        cout<<"\n Enter "<<i+1<<"th data=";
        cin>>data[i];
```

```

        i++;
    }
}
template<class T>
void Search_Array<T>::searchData(T S_data)
{
    for(i=0;i<N_data;i++)
    {
        if(S_data == data[i])
        {
            cout<<"\n"<<S_data<<" is avaialble at "<<i+1<<"th
position";
            flag=1;
        }
    }
    if(flag==0)
        cout<<"\n"<<S_data<<" isnot available in the array";
}

int main()
{
    try{// try block
        Search_Array<int> sA1(120);

        sA1.readArray();
        sA1.searchData(20);
    }
    catch(intexcep){// catch block
        cout<<"\n Exception: Input value is larger than the Array Size";
    }

    getch();
    return(0);
}

```

Output of the program:

Exception: Input value is larger than the Array Size";

CHECK YOUR PROGRESS

1. Choose the correct option

- (a) An exception is a _____.
(i) Compile time error
(ii) Run time error
(iii) Syntax error
(iv) None of the above
- (b) Which of the following is an example of Synchronous exception?
(i) Dividing any number by zero
(ii) Runtime errors occurred due to the overflow and underflow conditions
(iii) Unexpected situation due to the failure of a hardware component
(iv) Both (i) and (ii)
- (c) Which of the following is not an exception handling construct in C++ programming language?
(i) try
(ii) catch
(iii) terminate
(iv) throw
- (d) A ____ block in a program is a group of programming statements that may raise exceptions during program execution.
(i) try
(ii) catch
(iii) throw
(iv) None of the above
- (e) An exception handler is located in a _____ block.
(i) try
(ii) catch
(iii) throw
(iv) None of the above

- (f) The ____ construct is used to throw exceptions.
- (i) try
 - (ii) catch
 - (iii) throw
 - (iv) None of the above
- (g) We can use ____ to specify that no exception will be raised in a function.
- (i) throw(0)
 - (ii) throw(NULL)
 - (iii) throw("No Exception")
 - (iv) throw()
- (h) It is not recommended to raise exception from ____.
- (i) Constructor
 - (ii) Destructor
 - (iii) Friend function
 - (iv) None of the above
- (i) Which of the library function is used to install a function to handle an uncaught exception?
- (i) terminate()
 - (ii) unexpected()
 - (iii) set_terminate()
 - (iv) None of the above
- (j) Which of the library function is used to register a function to handle an unspecified exception?
- (i) set_unexpected()
 - (ii) unexpected()
 - (iii) set_default()
 - (iv) None of the above

4.9 SUMMING UP

- An exception is an anomaly or an unexpected condition or an error that can be occurred at the run-time of a program and it can change the normal execution flow of that program.

- Synchronous exceptions are raised due to any inappropriate input data or due to the use of any method in the program that is not appropriate for a new set of input data.
- Asynchronous exceptions are raised due to some external events that are not controlled by the program.
- Exception handling mechanism is used to catch exceptions and handle them before their occurrence so that they cannot disrupt the normal execution flow of the corresponding program.
- Exception handler is the part of a program written to handle probable exceptions. The job of an Exception handler is to catch the exceptions occurred at the time of program execution and then provides possible solutions to the exceptions or if required, terminates the program execution in a graceful manner.
- Exception handling mechanism helps to avert system failures and crashes. Exception handling mechanism separates the code to handle errors from the normal code in a program. Exception handling mechanism can be used to provide appropriate error messages to detect errors occurred in a system. It can also be used to confirm appropriate resource management in a system. Group of related errors may be handled by applying a single Exception handler.
- In C++, the Exception handling model is composed of three exception handling constructs. These three constructs are try, throw and catch.
- A try block in a program is a group of programming statements that may raise run-time errors or exceptions.
- The throw construct is used inside functions and try blocks to throw exceptions if runtime-errors are occurred during execution of the programming statements available in a function or a try block.
- In C++, an exception handler can rethrow a caught exception by invoking the throw construct without any arguments.

- The catch block in the exception handling mechanism is responsible for providing the exception handler for matching exceptions.
- When an exception is occurred in a constructor then the corresponding object may not be created fully and then the destructor is invoked to release the partially allocated resources. In inheritance, when an exception is raised in a base class constructor then constructors of the corresponding derived classes will not be invoked. If exceptions are raised in the constructors of both the base class and its derived class then the exception handler of the derived class will be executed before the exception handler of the base class.
- Exception handling in constructors provides proper release of resource allocations if any unexpected condition or run-time error occurs during the execution of a constructor.
- Throwing exception from destructor is not recommended as it may lead to some undesirable situations. But if it is required to throw exception from a destructor then the exceptions should be caught and handled inside the destructor itself.
- In C++, 'unexpected()', 'set_unexpected()', 'terminate ()' and 'set_terminate()' are the built-in functions used to handle uncaught exceptions..
- In C++, exception handling in class template is same with the normal exception handling.

4.10 ANSWERS TO CHECK YOUR PROGRESS

1.
 - (a). (ii) Run time error
 - (b). (iv) Both (i) and (ii)
 - (c). (iii) terminate
 - (d). (i) try
 - (e). (ii) catch
 - (f). (iii) throw
 - (g). (iv) throw()
 - (h). (ii) Destructor

- (i). (iii) `set_terminate()`
- (j). (i) `set_unexpected()`

4.11 POSSIBLE QUESTIONS

- 1) What is Exception? How Exceptions are handled in C++?
- 2) Write down the advantages of Exception handling mechanism.
- 3) Explain the Exception handling model in C++ programming language.
- 4) How uncaught exceptions are handled in C++?
- 5) What happens when exceptions are raised in constructors?
- 6) Why it is not recommended to raise exceptions from destructor?

4.12 REFERENCES AND SUGGESTED READINGS

- 1) Venugopal, K. R., Rajkumar, Ravishankar, T. *Mastering C++*. Tata McGraw-Hill Education, 2001.
- 2) Balagurusamy, E. *Object Oriented Programming with C++*. Tata McGraw-Hill, 2006
- 3) Jana, Debasish. *Java and object-oriented programming paradigm*. PHI Learning Pvt. Ltd., 2005.
- 4) Schildt, Herbert. *Java: the complete reference*. McGraw-Hill Education Group, 2014.

---x---

UNIT 1: INTRODUCTION TO FUNCTIONAL PROGRAMMING LANGUAGES

UNIT STRUCTURE

- 1.1 Introduction
- 1.2 Objectives
- 1.3 Principles of Functional Programming
 - 1.3.1 Function Prototype
 - 1.3.2 Function Definition
 - 1.3.3 Function call
- 1.4 Function parameters
- 1.5 Scope and environment of a variable
- 1.6 Recursive Functions
- 1.7 Virtual Functions
- 1.8 Introduction to LISP
 - 1.8.1 Debugging in CL
 - 1.8.2 Developing Programs in CL
 - 1.8.3 Functions in CL
 - 1.8.4 Named Functions
- 1.9 Summing up
- 1.10 Answer To Check Your Progress
- 1.11 Possible Questions
- 1.12 References and Suggested readings

1.1 INTRODUCTION

It is difficult to implement a large program although all the algorithms are available. To implement those types of program, program must be split into a number of subprograms with the help of functions. In this unit we will discuss about the principles of functional programming and its components. Here we will also discuss different parameter passing mechanism of a function in C++ and implementation of virtual functions. Finally we will discuss the

concept of LISP (LISt Processing) and implementation of it in different situations

1.2 OBJECTIVES

After going through this unit learner will able to

- Understand the concept of Functional programming.
- Learn about different parameter passing mechanisms in C++.
- Understand the concept of scope of a variable in a function.
- Learn about the recursive functions.
- Understand the concept of LISP and its implementation

1.3 PRINCIPLES OF FUNCTIONAL PROGRAMMING

A set of program that can be processed independently is known as function. In a large program a repeated group of instructions can arrange as a function. Functions are independent because the variable names within the function are local to that function. Parameters plays a vital role between the calling and called function. The use of functions makes easier to implement a complex program. The advantages of functional programming are as follows

- Modular programming
- Easy to debug
- Reducing the amount of work
- Due to the code reusability, size of the program is reduced
- Divide-and-conquer principle

Different components are required for constructing or working with functions. Every function has the following components

- Function prototype or function declaration
- Function parameter
- Function definition
- Function call
- Return statement

The Figure 1.1 shows the structure of a function

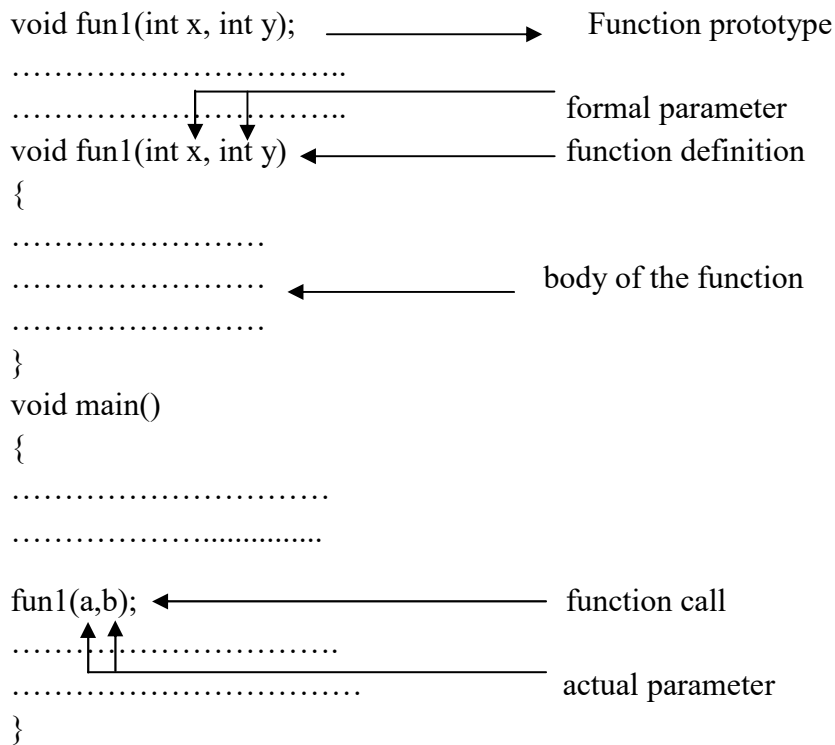


Figure 1.1 components of a function

1.3.1 Function Prototype

Function prototype provides the following information to the compiler

- The name of the function
- The type of the returned value (by default the returned value is integer).
- The types of arguments that must be supplied in a call to the function.

Prototype is the key components added to the C++ function. When a function call is encountered, the compiler checks the function call with the function prototype for the correct use of the arguments. If any violation is found, compiler will inform the user about that violence.

A function prototype is a declaration statement which has the following syntax

```

return_value function_name(argument1,argument2,.....
                           ,argument n);

```

The `return_value` specifies the data-type of the value in the return statement. The function can return any data-type and if there is no any return value it can be placed with “void” before the function name.

1.3.2 Function Definition

The function itself is referred to as function definition. The first line of the function definition is known as function declarator and it is followed by the function body. The body of the function is enclosed in braces, C++ allows the definition to be placed anywhere in the program. The prototype declaration is optional, if the function defined before its invocation. The definition of a function can be defined as follows

```
int big(int p, int q)  ← function declarator
{
    if (p > q)
    return p;  ← function body
    else
    return q;
}
```

1.3.3 Function call

A function call is specified by the function name followed by the arguments which is enclosed in parentheses and it will be terminated by a semicolon. A function can be call as follows

```
r = big (x,y)
```

The call statement causes the control to be transferred to the first statement in the function body and after execution of the function body the control is returned to the statement following the function call.

1.4 FUNCTION PARAMETERS

The parameter in the function call is known as the actual parameters and the parameters in the function declarator are known as the formal parameters.

`r = big(x,y)`

Here passes the parameter x and y to big(). The parameters p and q are formal parameters. When a function call is made, it will establish a communication between actual and formal parameters. The value of the variable x is assigned to p and the value of the variable y is assigned to q. Functions can be categorized into two categories one is function that do not have return value and the other is function with return value.

Here the return value of the function big() is assigned to the local variable “r” in the main() function. The Figure 1.2 shows the function big () returning a value to the caller.

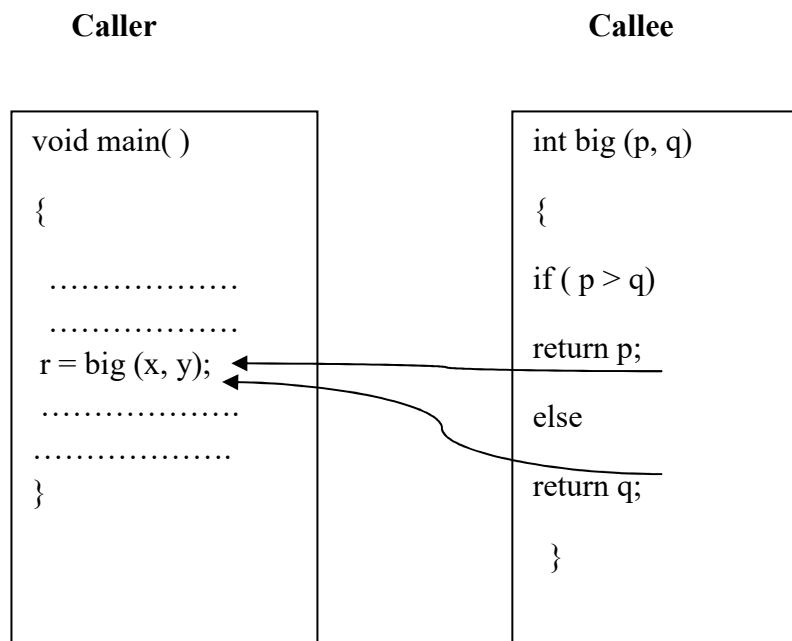


Figure 1.2 Function returning a value

The process of passing multiple parameters is similar to passing a single parameter. The value of the first actual parameter in the caller is assigned to the first formal parameter in the called function and the value of the second actual parameter is assigned to the second formal parameter in the called function.

1.4.1 Parameter Passing

It is a mechanism for communication of data and information between caller and called functions. C++ supports three types of parameter passing mechanism

- Pass by value
- Pass by Address
- Pass by reference(only in C++)

In case of parameter passing the following conditions must be satisfied for a function call

- The number of parameters in the function call and in the function declaration must be the same
- The data type of the each parameter in the function call must be the same with the corresponding parameter in the function declaration.

- **Pass by value**

Pass by value is the default mechanism of parameter passing technique. Pass-by-value mechanism does not change the contents of the argument in the calling function even if these arguments changes in the called function. Because the content of the actual parameter in a caller function is copied to the formal parameter in the callee or called function.

- **Pass by Address**

In pass by address mechanism the address of the variable is passed instead of passing the values of the variables. Here the address of the argument is copied into a memory location. The de-referencing operator is used in the called function for this purpose.

- **Pass by Reference**

Pass by reference has the functionality of pass-by-pointer but it uses the syntax of call by value. In pass by reference the function body and the call mechanism is similar to that of a call by value but it is actually a call by pointer. In the function declaration parameters which are to be received by reference must be preceded by the “&” operator. Here any modification done to these operators will also be reflected in the actual parameters.

The following points are important about the reference parameters

- Reference parameters always refer to variable and it can never be null.
- A reference can never be changed once it is established.
- There is no any explicit requirement to difference the memory address and accessing the actual value for a reference.

1.4.2 Default arguments

A function call should specify all the arguments used in the function definition. In case of C++, if one or more arguments are omitted the function take the default values for the omitted arguments. It will be done by providing the default values in the function prototype. The parameters without default value are placed first and those with default values are placed later. To establish a default value, the function prototype or the function definition must be used. The compiler checks the function prototype and declarator with the arguments in the function call and then it will provide default values to those arguments. The default arguments must be known to the compiler prior to the invocation of a function. It will reduce the complexity of passing arguments at point of function call.

1.5 SCOPE AND ENVIRONMENT OF A VARIABLE

In a program, every variable has some memory associated with it. Allocation of memory for a variable is released at different situations in the program. For example the allocation memory for the local variable is defined within a function is released when the function starts execution and released as the function return a value. A variable declared outside the all the function is known as the global variable. It will be accessed in the entire life-span of the program. The period of time during which the memory is associated with the variable is known as the extent of the variable. Consider the following function

```
void function1()  
{  
    int i;  
    i=5;  
}
```

Declaring the variable “i” as integer means deciding the memory location occupied by the variable “i”. The memory for this type of local variable is allocated in the program stack when the function is invoked. Here the memory allocated for “i” is released when the function is exit or terminated and the memory space is available for future use. Identifier defined in a function is not accessible outside the function that means their extent is limited of that function. Let us take the following example

```
void function1()  
{  
    int i;  
    i = 5;  
}  
  
void main()  
{
```

```

i = 10

function1 ( );

i = 20;

}

```

When the program is compiled, the statements `i=10` and `i=20` will lead a compilation errors. Here “i” is not visible inside the `main()` function. The identifier “i” is only valid inside the `function1()`. The region of a source code where the identifier is visible is known as the scope of the identifier. In the above program the scope of the variable “i” is within the `function1()` only.

1.6 RECURSIVE FUNCTIONS

Scientific operations may also be expressed using recurrence relations. In C++ allows the programmers to express these types of relations in special function is known as the recursive functions. Recursive function is a function that contains a function call to itself. The recursive function for computing the factorial of a number can be expressed as follows

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{fact}(n - 1), & \text{otherwise} \end{cases}$$

Recursion revolves around a function recalling itself. In a recursive function there must be one function call to itself. Two important conditions which must be satisfied by any recursive function are as follows

- Each time a function calls to itself it must be nearer to solution of the problem.
- There must a condition for stopping the process of computation.

1.7 VIRTUAL FUNCTIONS

In C++, polymorphism refers the form of a member function that can be changed at runtime. The function that can be changed at the runtime is known as the virtual function and the class that contains

the virtual function is known as the polymorphic class. The objects of the polymorphic class are addressed by the pointer and respond differently for the same message at the runtime. Types of polymorphism in C++ are shown in the Figure 1.3.

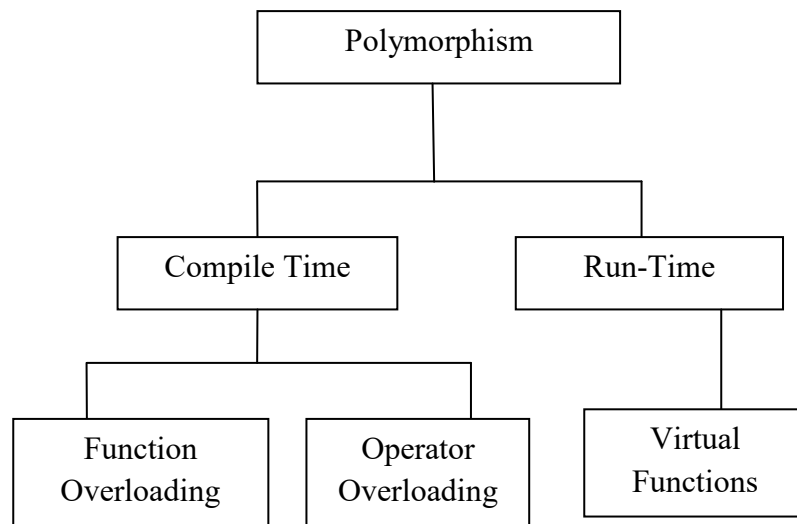


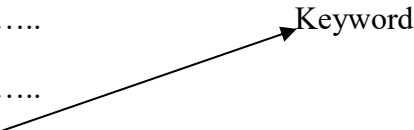
Figure 1.3. Types of polymorphism in C++

To realizing the polymorphism, function overloading and operator overloading features are implemented in C++. Polymorphism can also be implemented in C++ through dynamic binding mechanism. Compile time polymorphism can be achieved by two ways one is function overloading and other is operator overloading. Function overloading can be implemented by invoking function whose signature is similar with the arguments specified in the function call statement. Again the operator overloading is implemented by allowing operators to operate on some user-defined data-type with the same interface as that of the standard data types. Function call can be bound to the actual function either at the compile time or at runtime. Resolving a function call at compile time is known as static binding and resolving a function call at run time is known as dynamic binding. In C++, dynamic binding is achieved by using the virtual function. It allows programmer to declare functions in a base class which can be defined in the each derived class. The syntax of defining a virtual function in a class is shown as follows

```

class student
{
    public:
    .....
    .....
    virtual Return Type Function Name(arguments)
    {
        .....
        .....
    }
    .....
};

```



The keyword “virtual” provides a mechanism for defining the virtual function. The virtual keyword is used with the functions which are to be bound dynamically when we declaring the member function in the base class. It is recommended that a virtual function should be defined in the public section of a class. When virtual function is declared in the public section of a class it allows deciding which function to be used at runtime. In most of the cases virtual functions are defined with a null-body that means it has no definition. In such situation the functions in the base class are similar to *do-nothing* or *dummy* functions. These functions are called *pure virtual* function. The syntax of defining pure virtual function is as follows

```

class student
{
    public:
    .....
    .....
    virtual Return Type Function Name (arguments) = 0;
    .....
}

```

.....
};

A class containing pure virtual functions cannot be used to define any objects. Such type of classes is called abstract class.

Check Your Progress-I

1. Multiple Choice Questions

- (i) A set of program that can be processed independently is known as
- (a) Array
 - (b) String
 - (c) Function
 - (d) Structure
- (ii) In C++, the first line of the function definition is known as
- (a) function declaratory
 - (b) Function prototype
 - (c) First line
 - (d) Function heading
- (iii) The parameter in the function call is known as the
- (a) Formal parameter
 - (b) Actual parameter
 - (c) Default parameter
 - (d) Null parameter
- (iv) Default mechanism of parameter passing technique is known as
- (a) Call by address
 - (b) Call by reference
 - (c) Call by array
 - (d) Call by value
- (v) A variable declared outside the all the function is known as
- (a) Global variable
 - (b) Local variable
 - (c) default variable
 - (d) Null variable

2. State True or False

- (i) C++ allows the function definition to be placed anywhere in the program.
- (ii) The process of passing multiple parameters is different to passing a single parameter.
- (iii) A reference can never be changed once it is established.
- (iv) The class that contains the virtual function is known as the polymorphic class.
- (v) Resolving a function call at compile time is known as dynamic binding.

1.8 INTRODUCTION TO LISP

LISP high level programming language after FORTRAN. Lisp was invented by John McCarthy in 1958 and it was first implemented by Steve Russell on an IBM 704 computer. CL(Common Lisp) originated during the 1980 and 1990s. LISP serves as common language and it can be easily extended for specific implementation. It depends on the machine specific characteristics.

Some of the features of common LISP are as follows

- It is machine-independent
- It allows programs dynamically
- It provides high level debugging and advanced object-oriented programming.
- It provides a complete I/O library and extensive control structures.
- Automatic memory management/ Garbage collection
- Portability: CL programs are especially portable.
- Standard Features: CL contains many built-in features. Features such as full-blown symbol tables and package systems, hash tables etc. which are required in significant development of other languages.

CL is different from other languages due to its unique syntax and development mode. It allows programmer to make changes and test the codes immediately in an incremental manner. A more powerful

way to developing with CL is running it in a powerful text editors. One of the popular text editors for running a CL program is GNU Emacs. One advantages of this editor is that the editor keeps a history of the entire session. All items can be easily recalled according to the requirement of the programs.

Stop To Consider

GNU Emacs comes pre-installed with all Linux systems. Emacs distributions and documentation are available from <http://www.gnu.org>

To start Emacs, type the following lines

emacs

or

gnuemacs

then start a Unix shell within emacs by typing M-x shell. Now type

lisp

inside this shell's window to start a CL session .

To shut down a session, exit CL by typing

(excl:exit)

Exit from the shell by typing

exit

Implementation of CL supports IDE. This will provide a visual layout and some design capabilities for the user interface for application development.

1.8.1 Debugging in CL

CL provides an advanced debugging and error-handling capabilities. For this reason CL programs are very rarely crash in a fatal manner.

A break occurs when CL programs detected any errors. It will print an error message and presents the user with one or more possible restart actions. The debugger is itself a CL Command Prompt but it has some additional functionality. Some of the common debugger commands are as follows

reset→ returns to the top level

pop→ returns one level up

continue→ continues execution after a break or an error.

zoom→ Views the current Function call

up→ Moves current pointer up one frame in the stack

down→ Moves the current pointer down one frame in the stack

help→ Describes all the debugger commands

1.8.2 Developing Programs in CL

The development session includes the text editor with appropriate customizations and a CL process with any standard code loaded into it. While any programmer certainly developed an application for the top base of CL, then the programmer must be worked in the layered fashion. In top of the layer consists of some standard tools or libraries. In general CL packages and Macros must be defined before they can be used and referenced. One of the strong points of CL is its flexibility in working with the other environments. The following are the different ways in which CL can integrate with other language and environments.

- (i) Interfacing with the Operating System
- (ii) Foreign Function Interface
- (iii) Interfacing with Corba
- (iv) Interfacing with Windows (COM, DLL, DDE)
- (v) Code generation into other languages

CL uses the generalized prefix notation. One of the frequent actions in a CL program is call a function. In most of the cases it is done by writing an expression with names of the function, followed by its arguments. Let us take the following example

`(+ 2 2)`

In the above example the function name followed by the symbol “+”, followed by the arguments of two 2s. When the expression is evaluated it returns the value 4. One of the important things about LISP is the simplicity and consistency of its syntax. CL supports a full range of floating-point decimal numbers and true ratios. As we get the numbers the native data type which will simply evaluate to them when entered at the top level in an expression. Function arguments of a CL are evaluated in order from left to right and in the final stage they are passed to the function for the evaluation.

1.8.3 Functions in CL

Function is the basic building block of CL. The way to defining the named functions in CL is with macro `defun` (Definition of a function). Once the function has been defined with `defun`, programmer can call it like other function by writing it in parentheses with its arguments.

1.8.4 Named Functions

It is important to understand that a function is actually a kind of object separated from any symbol with it must be associated. Programmer can call actual Function-object in a symbol’s function-slot. This process is shown as following

`(symbol-function '+)`

`#<Function +>`

Functional arguments themselves are Function-objects in CL. The optional arguments to a function must be specified after any required arguments. The optional argument is identified by the symbol “&optional” in the argument list. Each optional argument can be either a symbol or a list of containing a symbol and an expression which will return a default values. In case of symbol the default value is taken to be NIL.

Check Your Progress-II

3. State True or False

- (i) LISP is serves as common language and it can be easily extended for specific implementation.
- (ii) LISP does not allow programmer to make changes and test the codes.
- (iii) A break occurs when CL programs detected any errors.
- (iv) CL is not flexible in working with the other environments.
- (v) CL implementation supports IDE.

1.9 SUMMING UP

- A set of program that can be processed independently is known as function and it is independent because the variable names within the function are local to that function.
- Prototype is the key components added to the C++ function. When a function call is encountered, the compiler checks the function call with the function prototype for the correct use of the arguments.
- The first line of the function definition is known as function declarator and it is followed by the function body.

- A function call is specified by the function name followed by the arguments which is enclosed in parentheses and it will be terminated by a semicolon.
- The parameter in the function call is known as the actual parameters and the parameters in the function declarator are known as the formal parameters.
- Functions can be categorized into two categories one is function that do not have return value and the other is function with return value.
- The process of passing multiple parameters is similar to passing a single parameter.
- Pass by value is the default mechanism of parameter passing technique.
- Pass-by-value mechanism does not change the contents of the argument in the calling function even if the arguments changes in the called function
- Global variable will be accessed in the entire life-span of the program and the period of time during which the memory is associated with the variable is known as the extent of the variable.
- A recursive function is function that contains a function call to itself.
- The function that can be changed at the runtime is known as the virtual function and the class that contains the virtual function is known as the polymorphic class.
- Polymorphism can be implemented in C++ through dynamic binding mechanism.
- Compile time polymorphism can be achieved by two ways one is function overloading and other is operator overloading.

- .Resolving a function call at compile time is known as static binding and resolving a function call at run time is known as dynamic binding.
- LISP high level programming language after FORTRAN and it was invented by John McCarthy in 1958.
- CL is different from other languages due to its unique syntax and development mode and it allows programmer to make changes and test the codes immediately in an incremental manner.
- While any programmer certainly developed an application for the top base of CL, then the programmer must be worked in the layered fashion.
- The way to defining the named functions in CL is with macro defun and once the function has been defined with defun, programmer can call it like other function by writing it in parentheses with its arguments.
- Functional arguments themselves are Function-objects in CL. The optional arguments to a function must be specified after any required arguments.
- The optional argument is identified by the symbol “&optional” in the argument list.

1.10 ANSWER TO CHECK YOUR PROGRESS

1.(i) (C)	(ii) (a)	(iii) (b)	(iv)(d)	(v)(a)
2. (i) True	(ii) False	(iii) True	(iv) True	(v) False
3.(i) True	(ii) False	(iii) True	(iv)False	(v)True

1.11 POSSIBLE QUESTIONS

1. What are the advantages of functional programming?
2. Explain the components of a function?
3. What is function prototype?

4. Differentiate between function definition and function declaration
5. Explain the different types of parameters in a function.
6. What is parameter passing in a function? Explain the different ways of parameter passing mechanism.
7. What is a scope of a variable? Explain with a suitable example.
8. What is recursive function?
9. What is virtual function? Explain its syntax.
10. What is dynamic binding? How is it achieved?
11. What is LISP? Explain its features.
12. How is function implemented in LISP?

1.12 REFERENCES AND SUGGESTED READINGS

- Venugopal, K. R., Buyya, R., & Ravishankar, T. (2010). Mastering C++. Tata McGraw Hill Education Private Limited.
- T.W. Pratt and M. V. Zelkowitz: Programming Languages: Design and Implementation; PHI.
- Ravi Sathi, Programming Languages, Concepts and Constructs, Pearson Education, Asia, LPE
- B. Stroustrup, The C++ Programming Language, Addison Wesley Publishing Company, 1995.
- Cooper Jr, D. J. (2003). Basic Lisp Techniques.

---X---

UNIT 2: FUNCTIONAL PROGRAMMING IN C++

Unit Structure:

- 2.1 Introduction
- 2.2 Objectives
- 2.3 Structure and Components of Function in C++
 - 2.3.1 Function Declaration
 - 2.3.2 Function Definition
 - 2.3.3 Function Call
 - 2.3.4 Parameters
- 2.4 Polymorphic Function in C++
 - 2.4.1 Function Overloading in C++
 - 2.4.2 Operator Overloading in C++
 - 2.4.3 Function Overriding in C++
- 2.5 Inline Function
- 2.6 Recursive Function in C++
- 2.7 Summing Up
- 2.8 Answers to Check Your Progress
- 2.9 Possible Questions
- 2.10 References and Suggested Readings

2.1 INTRODUCTION

We have already learnt about functional programming in the earlier unit. A function in C++ is a group of C++ statements that will perform a specific computational task. Different library functions are already available in C++ library. In C++, programmers can also define own functions as per their requirements. A C++ program can be developed to solve a computational problem by using different library functions and writing multiple user-defined functions as per requirements where each function will perform a specific job. In this unit, functional programming in C++ will be discussed.

2.2 OBJECTIVES

After going through this unit, we will be able to learn:

- About the structure and components of functions in C++.
- About polymorphic functions in C++.
- About Inline function in C++.
- About Recursive function in C++.

2.3 STRUCTURE AND COMPONENTS OF FUNCTION IN C++

The structure of a function in C++ is composed of the following components.

- Function Declaration
- Function Definition
- Function Call
- Parameters

Let us consider the following C++ program to understand the structure of function in C++ programming.

Program 2.1: C++ program with a function that will display all the prime numbers available within an input range and the function will return the total number of prime numbers that are displayed.

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

```
int printPrimeNumbers(int, int); // Function Declaration
```

```
int main()
{
    int start, end, totalPrime;
    cout<<"\n Enter the start value of the range=";
    cin>>start;
    cout<<"\n Enter the end value of the range=";
    cin>>end;
    if( (start >= 1 ) && ( start < end ))
    {
```

```

        cout<<"\nPrime numbers within "<<start<<" and "<<end<<" :\n";

totalPrime = printPrimeNumbers(start, end); // Function Call
    cout<<"\n Total number of displayed Prime numbers= ";
    cout<<totalPrime;
}
else
    cout<<"\n Wrong Input";

    getch();
    return(10);
}

int printPrimeNumbers(int s, int e) //Function Definition
{
    int countPrime = 0, i, j, flag;
    for(i = s+1; i<e; i++)
    {
        j = 2;
        flag = 1;
        while(j <= i/2)
        {
            if(i%j == 0)
            {
                flag = 0;
                break;
            }
            j++;
        }
        if(flag==1)
        {
            cout<<"\t"<<i;
            countPrime++;
        }
    }

    return(countPrime);/*Returns total number of available
Prime numbers */
}

```

Output:

```
Enter the start value of the range= 2
Enter the end value of the range= 10
Prime numbers within 2 and 10 :
3      5      7
Total number of displayed Prime numbers= 3
```

2.3.1 Function Declaration

The function declaration is also referred as function prototype. The function declaration of a C++ function is a statement used to notify the compiler about the presence and format of the function in a C++ program. Before using a function in a C++ program, its declaration has to be written in that program. The function declaration of a function consists of the return type, function name and data-types of the parameters that will be passed into that function. If the function will not return any value then the return type of its function declaration will be 'void'.

For example: In **Program 2.1**, function declaration of the function, 'printPrimeNumbers(int, int)' is written as:

```
int printPrimeNumbers(int, int);
```

In this statement, the return type is 'int' and it means that the function will return an integer value. The function name is 'printPrimeNumbers'. Finally, the parameter type list includes two 'int' data types.

2.3.2 Function Definition

We have already learnt that each function performs a specific task. The function definition of a C++ function includes a block of C++ statements to perform its job. A Function Definition of a C++ function consists of the return type, the function name, parameters with their data types and a block of C++ statements. When a function is called then its block of statements is executed.

For example: In **Program 2.1**, the following function definition is used where 's' and 'e' are the parameters passed into the function, printPrimeNumbers(). So, two integer values will be passed into the function.

```
int printPrimeNumbers(int s, int e) //Function Definition
```

```

{
    int countPrime = 0, i, j, flag;
    for(i = s+1; i<e; i++)
    {
        j = 2;
        flag = 1;
        while(j <= i/2)
        {
            if(i%j == 0)
            {
                flag = 0;
                break;
            }
            j++;
        }
        if(flag==1)
        {
            cout<<"\t"<<i;
            countPrime++;
        }
    }
    return(countPrime); /*Returns total number of available
Prime numbers */
}

```

This function definition will display all the prime numbers available within the integer values passed through the parameters, ‘s’ and ‘e’. The above function definition also returns the total number of prime numbers that are displayed by it.

2.3.3 Function Call

Function call statement is used when a library or a user defined function is required to be executed in a C++ program to perform a specific task provided by that function. A function call statement includes the function name followed by the parentheses, ‘()’. If the function requires any data to perform its job then required data can be passed into the function through parameters. In the function call statement, parameters can be passed inside the parentheses, ‘()’.

For example: In **Program 2.1**, the function call statement is written as:

```
totalPrime = printPrimeNumbers(start, end);
```

In this statement, the function, 'printPrimeNumbers' is called and two integer values are passed into it through the parameters, 'start' and 'end'. When this function call statement is executed then the block of C++ statements of the user-defined function, 'printPrimeNumbers' will be executed and an integer value will be returned which will be stored in the variable, 'totalPrime'.

2.3.4 Parameters

A parameter passed into a C++ function is actually a variable to store the required data that will be used by the function to perform its job. Any number of parameters can be passed into a function as per requirement. Parameters in case of C++ function can be categorized into two types that are Actual parameters and Formal parameters. Parameters used in the function call statement are called as Actual parameters. On the other hand, parameters used in the function definition are called as Formal parameters.

For example: In **Program 2.1**, 'start' and 'end' are the Actual parameters as they are used in the function call statement. On the other hand, 's' and 'e' are the Formal parameters as they are used in the function definition.

2.4 POLYMORPHIC FUNCTION IN C++

We have already learnt about Polymorphism in an earlier chapter. In this unit, we are going to discuss the implementation of different types of polymorphic functions in C++. In C++, polymorphic functions can be implemented by function overloading, operator overloading and function overriding.

2.4.1 Function Overloading in C++

In C++, multiple functions with same function name can be defined in a program where functionalities of each function are different from each other. It is referred as function overloading in C++. In this process, both the number of parameters and type of the parameters cannot be same among the functions with same name. Let us consider the following C++ program (**Program 2.2**) to understand the implementation of function overloading in C++.

Program 2.2: C++ program to show Function Overloading

```
#include<iostream.h>
#include<conio.h>
// Function prototypes for the overloaded functions
int average(int,int,int);
float average(float, float, float);
int average(int [],int);

int main()
{
    int num1,num2,num3,avg;
    int arrNum[200],i, N;
    float rnum1,rnum2,rnum3,ravg;
    cout<<"\n Enter three Integer numbers::";
    cin>>num1>>num2>>num3;
    avg=average(num1,num2,num3); /*‘average()’ with three
                                ‘int’ type parameters will be called*/
    cout<<"\n Average of the three Integer numbers is"<<avg;
    cout<<"\n\n Enter three Real numbers::";
    cin>>rnum1>>rnum2>>rnum3;
    ravg=average(rnum1,rnum2,rnum3); /* ‘average()’ with
    three‘float’ type parameters will be called*/
    cout<<"\n Average of the three Real numbers is"<<ravg;
    cout<<"\n\n Enter the total number of data to be stored in the array=";
    cin>>N;
    if(N>200)
        cout<<"\n Wrong input";
    else
    {
        cout<<"\n Enter "<<N<<" number of integer numbers into the array::";
        for(i =0;i<N;i++)
        {
            cout<<"\nEnter "<<i+1<<"th data into the array=";
            cin>>arrNum[i];
        }
        avg=average(arrNum,N); /*‘average()’ with one integer
        array and one ‘int’ type parameter will be called*/
        cout<<"\n\n Average of the numbers stored in the array is"<<avg;
    }
    return(0);
}
```

```
}
```

```
int average(int N1,int N2,int N3)
{
return((N1+N2+N3)/3);
}
```

```
float average(float N1,float N2,float N3)
{
return((N1+N2+N3)/3);
}
```

```
int average(intarrNum[],int N)
{
int i,total=0;
for(i =0;i<N;i++)
{
total+=arrNum[i];
}
return( total/N);
}
```

Output of the program:

Enter three Integer numbers::34 67 87
Average of the three Integer numbers is=62

Enter three Real numbers::33.5 8.54 91.2
Average of the three Real numbers is=44.413334

Enter the total number of data to be stored in the array=5
Enter 5 number of integer numbers into the array::
Enter 1th data into the array=5
Enter 2th data into the array=8
Enter 3th data into the array=12
Enter 4th data into the array=56
Enter 5th data into the array=89

Average of the numbers stored in the array is=34

In **Program 2.2**, three functions with same name ('average') are defined and from the program, following points are observed.

- 'int average(int, int, int);' is the function prototype of the first function with function name, 'average'. Three integer numbers are passed into this function and it will return the average of these three integer numbers. So, to call this function, three 'int' type parameters must be passed into the function call.
- 'float average(float, float, float);' is the function prototype of the second function with function name, 'average'. Three real numbers are passed into this function and it will return the average of these three real numbers. So, to call this function, three 'float' type parameters must be passed into the function call.
- 'int average(int [], int);' is the function prototype of the third function with function name, 'average'. One integer array and one integer number are passed into this function. The integer number represents the total number of integer numbers stored in the integer array. This function will return the average of the integer numbers stored in the integer array. So, to call this function, two 'int' type parameters has to be passed into the function call where the first parameter will be an array.

2.4.2 Operator Overloading in C++

We have already learnt about operator overloading in an earlier unit. In this chapter, we are going to discuss the implementation of operator overloading in C++ programming. In C++ programming, additional operation can be defined for an operator and it is termed as operator overloading. Operator overloading can be implemented in C++ by defining special member functions or friend functions in a class.

In C++, following syntax is used to define a special member function for operator overloading where 'returnType' is the type of the data that will be returned from the operator overloading function, '*operator*' is a C++ keyword, '*operatorSymbol*' is the symbol that represent the operator and '[Parameter List]' is the list of parameters that may be passed into the function depending upon the requirement.

```

return Type operator operator Symbol ( [Parameter List])
{
    // C++ statements to define additional operation for the
operator
}

```

Let us consider the following C++ program to overload -- (Decrement) operator which is a unary operator. In case of overloading a unary operator using special member function, no parameter is required to be passed into the operator overloading function.

Program 2.3: C++ program to overload -- (Decrement) operator

```

#include <iostream.h>

class Decrement
{
private:
    int Num;
public:
    Decrement()
    {
        Num = 0 ;
    }
    void Input( );
    void Display( );
    void operator --( ); /* Function prototype of the special
member function for Operator overloading */
};

void Decrement :: Input( )
{
    cout<< "\n Enter an integer number = ";
    cin>>Num;
}

void Decrement :: Display( )

```

```

{
    cout<< "\n The value in Num = " <<Num;
}

void Decrement :: operator --( ) /* Special member function for
Operator overloading*/
{
    Num = Num -5;
}

int main( )
{
    Decrement D1;
    D1.Input();
    D1.Display();
    --D1;
    cout<<"\n After decrementing the object D1:\n";
    D1.Display();
    return(0);
}

```

Output of the above program:

```

Enter an integer number = 34
The value in Num = 34
After decrementing the object D1:
The value in Num = 29

```

In the above C++ program (**Program 2.3**), operator overloading is performed on the decrement operator (--). The decrement operator is overloaded to decrement the object of the class 'Decrement' where the value of the variable 'Num' associated with an object is decremented by 5.

Let us consider the following C++ program to overload Subtraction (-) operator using special member function. As Subtraction (-) operator is a binary operator, at least one parameter is required to be passed into the special member function for overloading the operator.

Program 2.4: C++ program to overload -(Subtraction) operator using special member function

```
# include <iostream.h>

class complexNumber
{
private:
float rl,img;
public:
void readComplex();
void displayComplex();
complexNumber operator -(complexNumber);
};

void complexNumber::readComplex( )
{
cout<< "\n Enter the Real part of the Complex Number = ";
cin>>rl;
cout<< "\n Enter the Imaginary part of the Complex Number = ";
cin>>img;
}

void complexNumber :: displayComplex( )
{
if(img==0)
cout<<rl<< " + 0i";
else
{
if(img<0 )
{
if(img== -1)
cout<<rl<< " - i";

else
cout<<rl<< " - " << -1*img<< "i";

}
else
{

```

```

if(img==1)
cout<<rl<< " + " << "i";
else
cout<<rl<< " + " <<img<< "i";
}
}
}

complexNumber complexNumber:: operator -(complexNumber CN)
{
complexNumber tempCN;
tempCN.rl = rl - CN.rl;
tempCN.img = img - CN.img;
return(tempCN );
}

int main( )
{
complexNumber compN1 , compN2 , compNSub;
cout<< "\n Enter the First Complex Number:";
compN1.readComplex( );
cout<< "\n Enter the Second Complex Number:";
compN2.readComplex( );
compNSub = compN1-compN2;
cout<< "\n First Complex Number is =";
compN1.displayComplex( );
cout<< "\n Second Complex Number is =";
compN2.displayComplex( );
cout<< "\n Subtraction of the two Complex Number =";
compNSub.displayComplex( );
return(0);
}

```

Output of the above program:

```

Enter the First Complex Number:
Enter the Real part of the Complex Number = 7
Enter the Imaginary part of the Complex Number = 9
Enter the Second Complex Number:
Enter the Real part of the Complex Number = 2
Enter the Imaginary part of the Complex Number = 3
First Complex Number is =7 + 9i

```

Second Complex Number is $=2 + 3i$

Subtraction of the two Complex Number $=5 + 6i$

In the above program (**Program 2.4**), Subtraction (-) operator is overloaded to perform subtraction operation between two objects of the class, 'complexNumber'. It is observed that each object of the class, 'complexNumber' represents a complex number. So, subtraction operation on two complex numbers is implemented by overloading Subtraction (-) operator in this program.

In C++, when a non-member function is declared as friend inside a particular class using 'friend' keyword then it is referred as a friend function to that class. A friend function to a class can access the private members of that class. Operator overloading can be implemented using friend function. In case of overloading unary operators using friend function, one parameter must be passed to the operator overloading function and in case of overloading binary operators, two parameters are required to be passed into the operator overloading function. The syntax of friend function declaration in a class is presented as follows.

friend returnType operator operatorSymbol([Parameter List]);

Let us consider the following C++ program to understand the operator overloading using friend function.

Program 2.5: C++ program to overload Subtraction (-) operator using friend function

```
#include <iostream.h>

class complexNumber
{
    private:
        float rl, img;
    public:
        void readComplex();
        void displayComplex();
    /* Friend function declaration to overload Subtraction (-)
       operator. */
```

```

        friend complexNumber operator -(complexNumber,complexNumber);
};

```

```

void complexNumber::readComplex( )
{
    cout<< "\n Enter the Real part of the Complex Number = ";
    cin>>r1;
    cout<< "\n Enter the Imaginary part of the Complex Number = ";
    cin>>img;
}

```

```

void complexNumber :: displayComplex( )
{
    if(img==0)

        cout<<r1<< " + 0i";
    else
    {
        if(img<0 )
        {

            if(img== -1)
                cout<<r1<< " - i";
            else
                cout<<r1<< " - " << -1*img<< "i";

        }
    }
    else
    {
        if(img==1)
            cout<<r1<< " + " << "i";
        else
            cout<<r1<< " + " <<img<< "i";

    }
}
}

```

//Friend function definition to overload Subtraction (-) operator.

```

complexNumber operator -(complexNumber CN1, complexNumber CN2)
{

```

```

        complexNumber tempCN;
        tempCN.rl = CN1.rl - CN2.rl;
        tempCN.img = CN1.img - CN2.img;
        return(tempCN );
    }

int main( )
{
    complexNumber compN1 , compN2 , compNSub;
    cout<< "\n Enter the First Complex Number:";
    compN1.readComplex( );
    cout<< "\n Enter the Second Complex Number:";
    compN2.readComplex( );
    compNSub = compN1-compN2;
    cout<< "\n First Complex Number is =";
    compN1.displayComplex( );
    cout<< "\n Second Complex Number is =";
    compN2.displayComplex( );
    cout<< "\n Subtraction of the two Complex Number =";
    compNSub.displayComplex( );
    return(0);
}

```

Output of the above program:

```

Enter the First Complex Number:
Enter the Real part of the Complex Number = 3
Enter the Imaginary part of the Complex Number = 7
Enter the Second Complex Number:
Enter the Real part of the Complex Number =8
Enter the Imaginary part of the Complex Number = 2
First Complex Number is =3 + 7i
Second Complex Number is = 8 + 2i
Subtraction of the two Complex Number = -5 + 5i

```

In the above program (**Program 2.5**), Subtraction (-) operator is overloaded using friend function to perform subtraction operation between two objects of the class, 'complexNumber'. It is observed that each object of the class, 'complexNumber' represents a complex number. So, subtraction operation on two complex numbers is implemented by overloading Subtraction (-) operator using friend

function in this program. It is also observed that two objects of the class, 'complexNumber' is required to pass as parameters into the friend function for overloading the Subtraction operator.

In C++, the following operators cannot be overloaded by using friend function.

1. ->
2. ()
3. []
4. =

In C++, all operators cannot be overloaded. Following operators cannot be overloaded in C++.

1. . (dot operator)
2. :: (Scope resolution operator)
3. ?:
4. *
5. sizeof()

2.4.3 Function Overriding in C++

In C++, a member function of a base class can be overridden by a member function of its derived class. It is referred as function overriding. In this process, the member function in the base class is declared as virtual function by using 'virtual' keyword in C++. A virtual function and the function which override it are exactly same in respect of their names, number of parameters and types of matching parameters. But both the function performs different tasks. Let us consider the following C++ program to understand the implementation of function overriding in C++.

Program 2.6: C++ program to show function overriding

```
#include<iostream.h>

class student
{
private:
    char sName[200],sAddress[300],sCourse[50];
    int rollNo;
public:
    virtual void inputInfo();
    virtual void displayInfo();
```

```

};

void student::inputInfo()
{
    cout<<"\n\n Enter Student Information::";
    cout<<"\n Enter student's name=";
    gets(sName);
    cout<<"\n Enter student's address=";
    gets(sAddress);
    cout<<"\n Enter the course of the student=";
    gets(sCourse);
    cout<<"\n Enter student's roll number=";
    cin>>rollNo;
}

void student::displayInfo()
{
    cout<<"\n\n Entered Student Information::";
    cout<<"\n Stduent Name=";
    cout<<sName;
    cout<<"\n Student Address=";
    cout<<sAddress;
    cout<<"\n The course of the Student=";
    cout<<sCourse;
    cout<<"\n Student Roll Number=";
    cout<<rollNo;
}

class examination: public student
{
private:
    int markPaper1, markPaper2, markPaper3, markPaper4;
    float markTotal, per;
public:
    void inputInfo();
    void displayInfo();
};

void examination::inputInfo()
{
    cout<<"\n\n Enter Marks obtained in the Examination::";

```

```

cout<<"\n Enter marks obtained in Paper1=";
cin>>markPaper1;
cout<<"\n Enter marks obtained in Paper2=";
cin>>markPaper2;
cout<<"\n Enter marks obtained in Paper3=";
cin>>markPaper3;
cout<<"\n Enter marks obtained in Paper4=";
cin>>markPaper4;
markTotal= markPaper1+markPaper2+markPaper3+markPaper4;
per=markTotal/4;
}

```

```

void examination::displayInfo()
{
cout<<"\n\n Student's Examination Information::";
cout<<"\n Marks obtained in Paper1=";
cout<<markPaper1;
cout<<"\n Marks obtained in Paper2=";
cout<<markPaper2;
cout<<"\n Marks obtained in Paper3=";
cout<<markPaper3;
cout<<"\n Marks obtained in Paper4=";
cout<<markPaper4;
cout<<"\n Total marks obtained=";
cout<<markTotal;
cout<<"\n Obtained Percentage=";
cout<<per;
}

```

```

int main()
{
student *S,St;
examination Ex;
S=&St;
S->inputInfo();
S->displayInfo();
S=&Ex;
S->inputInfo();
S->displayInfo();
return(0);
}

```

Output of the program:

Enter Student Information::
Enter student's name=Sarat Das
Enter student's address= Guwahati, Assam
Enter the course of the student=M.Sc.IT
Enter student's roll number=1

Entered Student Information::
Stduent Name=Sarat Das
Student Address=Guwahati, Assam
The course of the Student=M.Sc.IT
Student Roll Number=1

Enter Marks obtained in the Examination::
Enter marks obtained in Paper1=78
Enter marks obtained in Paper2=67
Enter marks obtained in Paper3=76
Enter marks obtained in Paper4=90

Student's Examination Information::
Marks obtained in Paper1=78
Marks obtained in Paper2=67
Marks obtained in Paper3=76
Marks obtained in Paper4=90
Total marks obtained=311
Obtained Percentage=77.75

In the above program (**Program 2.6**), 'student' is a base class and 'examination' is a derived class. 'student' contains two virtual functions that are 'virtual void inputInfo()' and 'virtual void displayInfo()'. Both these functions are overridden by 'void inputInfo()' and 'void displayInfo()' respectively that are defined in the derived class, 'examination'. It is observed that a pointer to 'student' class is used to refer an object of 'examination' class so that it can be used to override the virtual functions at runtime.

2.5 INLINE FUNCTION

We have already learnt about the advantages and disadvantages of writing user-defined functions in C++

programming. The main drawback of writing functions is that overhead will be involved in each function call due to the movement of the program control from the function call statement to the function statements and from the function to the function call statement. In C++, the concept of Inline function is provided as a solution to this problem. In case of an inline function, the function call is replaced with the actual programming statements of the function at compile time. As a result, the overhead associated with the function call will not occur. But use of Inline functions may increase the code size of a program. So, the concept of Inline functions should be used for the functions that contain small number of programming statements. In C++, 'inline' keyword is used to define an Inline function. The syntax of writing Inline function is presented as follows.

```
inline returnType function_Name([Parameter list])
{
    // C++ programming statements
}
```

Let us consider the following C++ program to understand the implementation of Inline function.

Program 2.7: C++ program to show the use of Inline function.

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

inline int square(int N)// Inline Function
{
    return(N*N);
}

int main()
{
    int edge;
    cout<<"\n Enter the length of an edge in a cube=";
    cin>>edge;
    cout<<"\n The surface area of the cube is"<< 6*square(edge);
    return(0);
}
```

Output of the program:

Enter the length of an edge in a cube= 2

The surface area of the cube is= 24

2.6 RECURSIVE FUNCTION IN C++

We already learnt that when a function contains a function call statement to call itself then this type of function is termed as Recursive function. Let us consider the following C++ program to understand the implementation of recursive function in C++ programming.

Program 2.7: C++ program to implement Binary Search algorithm using recursive function.

```
#include <iostream.h>

class binarySearch
{
    private:
        int data[200];
    public:
        void readData(int);
        void displayData(int);
        int bSearch(int,int,int);
};

void binarySearch::readData(int nData)
{
    int i;
    for(i=0;i<nData;i++)
    {
        cout<<"\n Enter "<<i+1<<"th data::";
        cin>>data[i];
    }
}

void binarySearch::displayData(int nData)
{
```

```

        int i;
        for(i =0;i<nData;i++)
        {
            cout<<"\t"<<data[i];
        }
    }

int binarySearch::bSearch(int startIndex, int endIndex, int srcData)
{

    int mid;

    if (startIndex>endIndex)
        return -1;

    else
    {
        int mid = startIndex+(endIndex-startIndex) / 2;

        if (data[mid] == srcData)
            return mid;

        if (data[mid] >srcData)
            return bSearch(startIndex, mid - 1, srcData); /* Recursive
            function call*/

        else
            return bSearch(mid + 1, endIndex, srcData); /* Recursive
            function call */
    }
}

int main()
{

    binarySearch B1;
    int N, index, srcData;

    cout<<"\n Enter the number of data to be stored in the array:.";
    cin>>N;

```

```

if(N>200)
cout<<"\n Wrong Input";
else
{
    B1.readData(N);
    cout<<"\n Enter the data to be searched=";
    cin>>srcData;
    cout<<"\n Elements in the array are::\n";
    B1.displayData(N);
    index = B1.bSearch(0, N-1, srcData);
    if (index == -1)
        cout<<"\n Searched data is not available in the array";
    else
        cout<<"\n Searched data is available in the array at the
subscript value "<<index;
    }

return 0;
}

```

Output of the program:

```

Enter the number of data to be stored in the array:: 6
Enter 1th data:: 12
Enter 2th data:: 45
Enter 3th data:: 78
Enter 4th data:: 90
Enter 5th data:: 167
Enter 6th data:: 201

```

Enter the data to be searched= 90

Elements in the array are::

12 45 78 90 167 201

Searched data is available in the array at the subscript value 3

In the above C++ program (Program 2.7), Binary search algorithm is implemented using the recursive function, 'int bSearch(int, int, int)'. This function is a member function of the class, 'binarySearch'.

CHECK YOUR PROGRESS

1. Choose the correct option

- (a) Which of the following is not a main component of C++ functions?
 - (i) Function Prototype
 - (ii) Parameters
 - (iii) Function call
 - (iv) None of the above

- (b) Which of the following is not related to polymorphic functions in C++ programming?
 - (i) Function overloading
 - (ii) virtual function
 - (iii) inline function
 - (iv) None of the above

- (c) In C++, operator overloading can be performed by using_____.
 - (i) Special member function
 - (ii) Friend function
 - (iii) virtual function
 - (iv) Both (i) and (ii)

- (d) Which of the following operator cannot be overloaded in C++ programming?
 - (i) ::
 - (ii) new
 - (iii) ++
 - (iv) %

- (e) If a function calls itself then it is termed as_____.
 - (i) Virtual function
 - (ii) Recursive function
 - (iii) Inline function
 - (iv) None of the above

2.7 SUMMING UP

- A function in C++ is composed of the following components.
 - Function Declaration
 - Function Definition
 - Function Call
 - Parameters
- The function declaration of a C++ function is a statement used to inform the compiler about the presence and format of the function in a C++ program.
- The function definition of a C++ function includes a group of C++ statements to perform its job.
- Function call statement is used when a library or a user defined function is required to be executed in a C++ program to perform a specific task provided by that function.
- A parameter passed into a C++ function is a variable to store the required data that will be used by the function to perform its job.
- Parameters used in the function call statement are called as Actual parameters and parameters used in the function definition are called as Formal parameters.
- In C++, polymorphic functions can be implemented by function overloading, operator overloading and function overriding.
- In C++, multiple functions with same function name can be defined in a program where functionalities of each function are different from each other.
- In C++, Operators can be overloaded by defining special member functions or friend functions in a class.
- In C++, a member function of a base class can be overridden by a member function of its derived class.
- In case of an inline function, the function call is replaced with the actual programming statements of the function at compile time.
- We already learnt that when a function contains a function call statement to call itself then this type of function is termed as Recursive function.

2.8 ANSWERS TO CHECK YOUR PROGRESS

1.

- (a) (iv) None of the above
- (b) (iii) inline function
- (c) (iv) Both (i) and (ii)
- (d) (i) ::
- (e) (ii) Recursive function

2.9 POSSIBLE QUESTIONS

1. Explain about the structure of functions in C++ programming.
2. How operators can be overloaded in C++ programming? Explain with examples.
3. Explain function overloading in C++ programming. Give example.
4. Explain function overriding in C++ programming. Give example.
5. What is Inline function? How Inline function can be useful in C++ programming?
6. What is Recursive function? Write a C++ program to demonstrate the use of Recursive function.

2.10 REFERENCES AND SUGGESTED READINGS

- 1) Venugopal, K. R., Rajkumar, Ravishankar, T. *Mastering C++*. Tata McGraw-Hill Education, 2001.
- 2) Balagurusamy, E. *Object Oriented Programming with C++*. Tata McGraw-Hill, 2006

---X---

UNIT 3: LOGIC PROGRAMMING LANGUAGES - I

Unit Structure:

- 3.1 Introduction
- 3.2 Unit Objectives
- 3.3 Review of Predicate Logic
 - 3.3.1 Proposition Calculus
 - 3.3.2 Predicate Calculus
- 3.4 Logic as a language for problem solving
- 3.5 Facts, rules, queries and deductions, sentence structure
- 3.6 General structure and computational behavior of logic programs
- 3.7 Summing Up
- 3.8 Answers to Check Your Progress
- 3.9 Possible Questions
- 3.10 References and Suggested Readings

3.1 INTRODUCTION

The base of logic programming is in structuring programs as sets of sentences expressed in symbolic logic. With its inherent design, logic programming efficiently handles queries by finding their truth value and giving choices that satisfies the query specifications.

Logic programming shows useful in natural language processing, database management, and predictive analysis—that are all gaining energy with global digital transformation. This makes logic programming an important language for many programmers.

3.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Understand Mathematical Logic as a basis for logic programming

- Explain the basic terms used in logic programs and its syntax and structure
- Analyze the computational behavior of logic programs

3.3 REVIEW OF PREDICATE LOGIC

One component of mathematical logic is **proposition calculus** which works with statements with values true and false and is therefore related with analysis of propositions. And the other part is **predicate calculus** which works with the predicates which are propositions having variables.

3.3.1 Proposition Calculus

A number of words making a complete grammatical structure having a sense and meaning and also mean an assertion in logic or mathematics is known as a sentence. This assertion can be of two types - declarative and non-declarative. A **Proposition** or **Statement** is a declarative sentence that is either true or false. For example, "Three plus three equals six." and "Three plus three equals seven." are both statements: the first because it is true and the second because it is false. Similarly, " $x + y > 1$ " is not a statement because for some values of x and y the sentence is true, but for others it is false. For example, if $x = 1$ and $y = 2$, the sentence is true, if $x = -3$ and $y = 1$, it is false. The truth or falsity of a statement is known as its **truth value**. Since only two possible truth values are accepted, therefore, this logic is sometimes called **two-valued logic**. Questions, exclamations and commands are not considered as propositions. For examples, let us consider the following sentences:

- (a) The sun rises in the north.
- (b) $2 + 4 = 6$
- (c) $(5, 6) \subset (7, 6, 5)$
- (d) Close the door.

The sentences (a), (b) and (c) are statements, the first is false and second and third are true.

(d) is a question, not a declarative sentence, hence it is not a statement.

It is usual to represent simple statements by letters p, q, r, \dots , called **proposition variables**. (Note that generally a real variable is represented by the symbol x . This means that x is not a real number but can take a real value. Similarly, proposition variable is not a proposition but can be replaced by a proposition). Propositional variables can only assume two values, true or false. There are also two **propositional constants**, T and F , that represent true and false, respectively. If p denotes the proposition "The sun sets in the west", then instead of saying the proposition "The sun sets in the north" is false, one can simply say the value of p is F .

A proposition consisting of only a single propositional variable or a single propositional constant is called an **atomic (primary, primitive)** proposition or simply proposition; that is they cannot be further subdivided. A proposition found from the grouping of two or more propositions by means of logical operators or connectives of two or more propositions or by negating a single proposition is referred to **molecular or composite or compound proposition**.

The words and phrases (or symbols) used to form compound propositions are known as connectives. There are five basic connectives called Negation, Conjunction, Disjunction, Implication or Conditional, and Equivalence or Biconditional. The following symbols are considered as connectives.

Symbol used	Connective words	Natural of the Compound statement formed by the connective	Symbolic form
$\neg, \sim, \bar{}$	not	Negation	$\sim p$
\wedge	And	Conjunction	$p \wedge q$
\vee	or	Disjunction	$p \vee q$
\Rightarrow, \supset	if...then	Implication (or Conditional)	$p \Rightarrow q$
$\Leftrightarrow, \leftrightarrow$	if and only if	Equivalence (or Bi-conditional)	$p \Leftrightarrow q$

3.3.2 Predicate Calculus

The propositional calculus does not let us represent many of the statements that we use in mathematics, computer science and in everyday life. In fact, Predicate calculus is a generalization of

propositional calculus. It includes all the components of propositional calculus- propositional variables and constants. Thus, Predicate calculus is important for several reasons-this has application in expert system, in database and also basis for the Prolog language.

Actually, a part of a declarative sentence describing the properties of an object or relation among objects is called a predicate. For example, consider two propositions:

Jadu is a bachelor.

Shyam is a bachelor.

Here, both Jadu and Shyam has the same property of being bachelor. In the propositional calculus we do not have symbolic representation of "is a bachelor" since this phrase, or predicate, is not a sentence. We can replace the two propositions by a single proposition 'x is a bachelor'. By replacing x by Jadu, Shaym (or by any other name), we can obtain many propositions. In logic, predicates can be derived by removing any nouns from a statement. Predicates are symbolised by a capital letter and the names of individuals or objects in general by small letters. Here, the sentence "x is a bachelor" is symbolised as $P(x)$, where x is a **predicate variable**. When concrete values are substituted in place of x (predicate Variable), a statement obtains. $P(x)$ is also known as a **propositional function**, because each choice of x produces a proposition $P(x)$ that is either true or false. Therefore, a **predicate** is a sentence that contains a finite number of variables and becomes a proposition when specific values are substituted for the variables. The **domain (universe of discourse or simply universe)** of a predicate variable is the set of all possible values that may be substituted in place of variables. For example, the domain for $P(x)$: "x is a bachelor", can be considered as the set of all human names.

Universal Quantifier-One clear way to change predicates into statements is to allocate specific values to their variables. Another way to get statements from predicates is to add quantifiers. Quantifiers are words that mean quantities such as some, few, many, all, none and indicate how often a certain statement is true.

The phrase "for all" (denoted by \forall) is called the universal quantifier. For example, consider the sentence "All human beings are mortal".

Let $P(x)$ represents "x is mortal".

Then the above sentence can be written as

$$(\forall x \in U) P(x) \text{ or } \forall x P(x) \dots (1)$$

where U is the Universe of discourse representing the set of all human beings. $\forall x$ represent each of the following phrases, since they have essentially the same

for all x
for every x
for each x

The statement (1) is known as a **universal statement**. Here, the expression $P(x)$ by itself is an open sentence and therefore has no truth value. However $\forall x P(x)$ does have a truth value and is assigned truth values as below:

$\forall x P(x)$ is true if $P(x)$ is true for every x in U ;

$\forall x P(x)$ is false, if and only if, $P(x)$ is false for at least one x in U .

A value for which $P(x)$ is false is said to be a **counter example** to the universal statement.

Existential Quantifier-The phrase “there exists” (denoted by \exists) is called the **existential quantifier**. For example, consider the sentence:

“There exists x such that $x^2 = 5$ ”. This sentence can be written as:

$$(\exists x \in R) P(x) \text{ or } \exists x P(x) \dots (2)$$

where $P(x)$ “ $x^2=5$ ”.

$\exists x$ represents each of the following phrases:

There exists an x
There is an x
For some x
There is at least one x

The statement (2) is called an **existential statement**. $\exists x P(x)$ has these truth values,

$\exists x P(x)$ is true if $P(x)$ is true for at least one x in U .

$\exists x P(x)$ is false if $P(x)$ is false for every x in U .

In particular, If $\{x: x \in P(x)\} \neq \Phi$ then $\exists x P(x)$ is true otherwise $\exists x P(x)$ is false.

When the quantifiers are used, one should specify the universe of discourse. If the universe of discourse is changed, the truth value may change. For example: $R(x): x^2 = 3$

If the universe of discourse is the set of all integers, then $\exists xR(x)$ is false. If the universe of discourse is the set of all real numbers, then $\exists \exists x R(x)$ is true.

Stop to Consider

Unlike propositional logic, which deals with simple true/false statements, predicate logic introduces predicates, variables, constants, and quantifiers. These elements help in modeling real-world problems that involve multiple objects and their interactions.

3.4 LOGIC AS A LANGUAGE FOR PROBLEM SOLVING

Logic is related with methods of reasoning. The Greek philosopher and scientist Aristotle (384–322 BC) is known as the first person to have studied logical reasoning. Logical reasoning is the soul of mathematics and is therefore an important starting point for study of discrete mathematics. Logic, among other things, has given theoretical basis for many areas of computer science such as digital logic design, automata theory, and computability and artificial intelligence etc.

The primary language for logic programming is Prolog, although there are many others, those have been implemented.

Key feature of Prolog is – it is "algorithm free" programming i.e. the programmer specifies what the output is supposed to be, NOT how to find it.

Application domains for logic programming

1. prototyping
2. natural language processing (parsing)
3. database querying
4. AI research
 - symbolic computation
 - theorem proving
 - expert system
5. parallel programming

3.5 FACTS, RULES, QUERY, DEDUCTION AND SENTENCE STRUCTURE

Relations-A relation is a function that returns Boolean value (also called a **predicate**)

A relation can be represented by a table of values that make the predicate true. For example, $<$ can be represented by:

0	1
0	2
.	.
.	.
.	.
1	2
1	3
.	.
.	.
.	.

Prolog uses relations completely - there are no other kinds of functions or procedures. On the other hand, any function can be changed to a relation by adding an extra parameter to represent the result. For example, consider an append function that takes two lists and returns a list (which is the concatenation of the two argument lists). Then the relation `append(A, B, C)` is true if A, B and C are lists such that appending A and B yields C. As tuples/rows in a table, `([1, 2], [2, 3], [1, 2, 2, 3])` is in `append`, while `([1,2],[2,3],[1,3])` is not.

Let us consider the following Prolog program, which states mother and father relations:

`mother (radhika, juhi).`

`mother (radhika, tarun).`

`mother(juhi, joey).`

`father (bijoy, juhi).`

`father(bijoy, tarun).`

`father (tarun, ani).`

For example, the rule `mother(radhika, juhi).` is meant to indicate that radhika is the mother of juhi. This code specifies the relations in the same way that tables would.

Given these definitions (i.e. after loading the file containing these definitions), the Prolog interpreter can answer yes/no questions about these relations. The **query**:

father(bijoy, tarun).

will cause the interpreter to return yes, while the query:

father (bijoy, ani).

will cause the interpreter to return no.

Note that queries are always typed in at the interpreter prompt - they can never be contained in files.

Rules-For the given mother and father relations previously specified, **rules** can be used to specify more interesting relations. For example:

parent(X, Y) :- mother(X, Y).

parent(X, Y) :- father (X, Y).

is a pair of rules that specifies that X is a parent of Y if X is the mother or father of Y. In Prolog, variables always start with uppercase letters, while identifiers beginning with lowercase letters represent data (atoms).

More example rules:

grandparent (X, Y) :- parent (X, Z) , parent(Z, Y).

ancestor (X, Y) :- parent(Z, Y),

ancestor(X, Y) :- parent(Z, Y), ancestor(X, Z)

The general form of a rule is:

$P :- Q_1, Q_2, \dots, Q_n$

Where P and Q_i can be references to relations (or variables or constants).

Logically, a rule of this form means :

$Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \Rightarrow P$

English interpretation : if Q_1, Q_2, \dots, Q_n are true, then P is true

Prolog interpretation: to prove P, it is sufficient to prove Q_1, Q_2, \dots, Q_n

This style of rule is known as a **Horn clause**, and is executed comparatively efficiently. A sequence of rules with the same L.H.S.:

$P :- Q_1, Q_2, \dots, Q_n$

$P :- K_1, K_2, \dots, K_m$

Is equivalent to :

$(Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \Rightarrow P) \vee (K_1 \wedge K_2 \wedge \dots \wedge K_m \Rightarrow P)$

Example :

parent (X, Y) :- mother (X, Y).

parent (X, Y) :- father(X, Y).

English: X is a parent of Y if X is the mother of Y or if X is the father of Y.

Example (list membership) :

mem(X, [Y | Z]) :- X = Y.

`mem(X, [Y | Z]) :- mem(X, Z).`

`[Y | Z]` is used in Prolog as list notation –where Y is the first element of the list, and Z is the rest of the list. So these two rules specify that X is a member of the list `[Y | Z]` if X is equal to Y (the first element), or X is member of the list Z. This is same as the usual recursive definition, except that there is no base case. Prolog uses the **Closed World Assumption** - if something can't be proven using the rules in the program, then it must be false. Hence the query:

`mem (X, []).`

will cause the interpreter to return no, because it can't be proven using the rules for mem above. (Note that `[]` is the empty list.)

A rule with no R.H.S., i.e.:

`P.`

is an assertion that P is true - nothing needs to be proven to show it. Such a rule is called a **fact**. Facts are often used for defining simple relations (see mother and father relations given above).

Example (list membership again):

`mem (X, [X,Z]).`

`mem (X, [Y|Z]) :- mem(X, Z).`

Note carefully the use of pattern matching in the fact.

Queries-A query is a question about a relation that is given to the Prolog interpreter. Recall that queries must be typed at the interpreter prompt - they can never be contained in files.

Given either of the previous definitions of mem, the query:

`mem(a, [2, b, a]).`

returns yes, meaning true or provable.

The query:

`mem(a, [2, b]).`

returns no, meaning false, fail or not provable.

If a query contains variables, Prolog will try to find values for the variables that make the query true (provable) For example, the query:

`mem (X, [2, b, a]).`

returns `X = 2`, which makes the query provable. Clicking the Next button (or entering a; in the Windows version) will cause Prolog to look for more solutions (more values of X that make the query true).

Entering the query:

`mem(2, X).`

and then clicking the Next button repeatedly (or entering ; repeatedly) will cause the sequence:

$X = [2 \mid _]$.
 $X = [_, 2 \mid _]$.
 $X = [_, _, 2 \mid _]$.
 $X = [_, _, _, 2 \mid _]$.

and so on. $_$ is a special anonymous (unnamed) variable with the property that any occurrence of $_$ is assumed to be a unique variable name. I.e.: $X = [_, _, _, 2 \mid _]$ does NOT mean that the first 3 elements of the list are required to all be the same.

Relations are more flexible than functions - some or all arguments can be variables (unspecified)

More Examples-A program to append lists:

`append ([], Y, Z).`
`append ([H | X], Y, Z) :- append (X, Y, Z2), Z = [H | Z2].`

Note that this is a recursive definition: appending the empty list with any list just gives that list (base case), while for a nonempty list, just append all of the list except the first element with the other list, and then stick the first element on the front of the result. The second rule can also be given as:

`append([H | X], Y, [H | Z]) :- append(X, Y, Z).` Again, queries can omit some or all arguments:

`append ([a, b], [c], X).`
 returns $X = [a, b, c]$.. while:
`append ([a, b], X, [a, b, c]).`
 returns $X = [c]$.

`append` can be used to define other interesting relations. A list A is a prefix of another list B if A can be appended with some other list to produce B:

`prefix(A, B) :- append (A, _, B) .`

The variable $_$ is used above because the value of that variable doesn't matter - only that some value for it exists. A list A is a suffix of another list B if some other list can be appended with A to produce B:

`suffix(A, B) :- append(_ , A, B).`

Relations/terms can be used to represent data structures. For example, binary search trees of numbers can be represented by two terms: empty and node (L, D, R), where L. is the left subtree, D is the (numeric) data and R is the right subtree. For this given representation, standard tree operations can be defined using rules:

`/* relation to check whether some value occurs in the tree */`
`isin(K, node(_, K, _)).`

```

isin(K, node (L, D,R)) :- K <D,isin (K, L).
isin(K, node (L, D,R)) :- K >D,isin (K, R).

/* adding a node at the proper position in the tree */
insert(K, empty, node (empty, K, empty)).
insert(K, node (L, D, R),node (L2, D, R)) :- K <D,insert (K, L, L2) .
insert(K, node (L, D, R),node (L, D, R2)):-K >D,insert (K, R, R2) .

```

Therefore in Prolog, deduction is obtained through a system of facts, rules, and queries that form the base of its sentence structure. Facts represent known truths, rules establish relationships between facts, and queries allowing asking questions that the system uses to deduce answers. The key to this process is the use of predicates and unification.

Unification-

Prolog uses a process known as unification to match patterns and find solutions. Unification tries to find bindings for variables in the query that make the query true based on the facts and rules.

Deduction-

When a query is asked, Prolog searches through its knowledge base (facts and rules) and uses unification to deduce the answer. If the query can be made true through a group of facts and rules, the interpreter deduces the solution.

Example:

Let us consider the following Prolog program:

```

father(ram, sanu).
mother(mala, sanu).
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).

```

If query ?- parent(ram, sanu). Prolog will:

1. Match the query with the head of the first rule parent(X, Y) :- father(X, Y)..
2. Unify X with ram and Y with sanu.
3. Check if father(ram, sanu) is a fact (which it is).
4. Return "true" because the query can be deduced to be true.

3.6 GENERAL STRUCTURE AND COMPUTATIONAL BEHAVIOR OF LOGIC PROGRAM

Prolog Syntax-The fundamental unit of Prolog syntax is the **term**. A **simple term** is a number, a variable or an **atom**- a string beginning with a lower case letter. A **compound term** is an atom followed by a parenthesized list of terms.
e.g.: mem (a, [a, b]).

In a compound term, the atom (mem in this example) is called a **functor**, and the parenthesized terms are called **arguments**.

In EBNF:

<term> --><sterm> | <cterm>

<sterm> --><variable> | <atom>|<number>

<cterm -->'<atom>'(<term> {, <term>}')

A rule is either a fact or a term to be concluded from other terms. A fact is just a term:

<rule> --><fact>|<term>:- <term> {, <term>}.

<fact> --><term>.

In the second form, the term on the L.H.S.is called the **head** or **goal** of the rule, and the terms on the R.H.S. are called **subgoals**_. A fact is just a goal.

A query is one more terms ending with a period.

<query>--><term>{, <term>}.

The terms in a query are also called goals.

Prolog programs gets executed through a process known as resolution, where a query is compared against rules and facts in the program. The Prolog engine attempts to find a solution by unifying the query with facts and rules, essentially creating a "proof" that the query is true. This process includes backtracking if a path to a solution isn't found, allowing the engine to explore alternative path.

Backtracking-If the goals of multiple rules match the current subgoal, each of those rules can be tried (in order). When backtracking occurs (because the current choice of a rule failed), the interpreter:

- goes back to the most recent point where it had a choice of rules
- restores the subgoal list and substitutions to their values at that point

- chooses the next rule
- continues executing

This form of backtracking is called chronological backtracking, because the most recent rule choice is always changed.

Applying rules and backtracking forms a search tree, where each branch in the tree results from backtracking. Prolog answers a query by performing a depth first search in this tree. In fact, the search is depth first because the backtracking is chronological.

For example, consider the rules for `isin` (numbered for reference):

/ * 1 */ `isin(K, node(_, K, _))`.

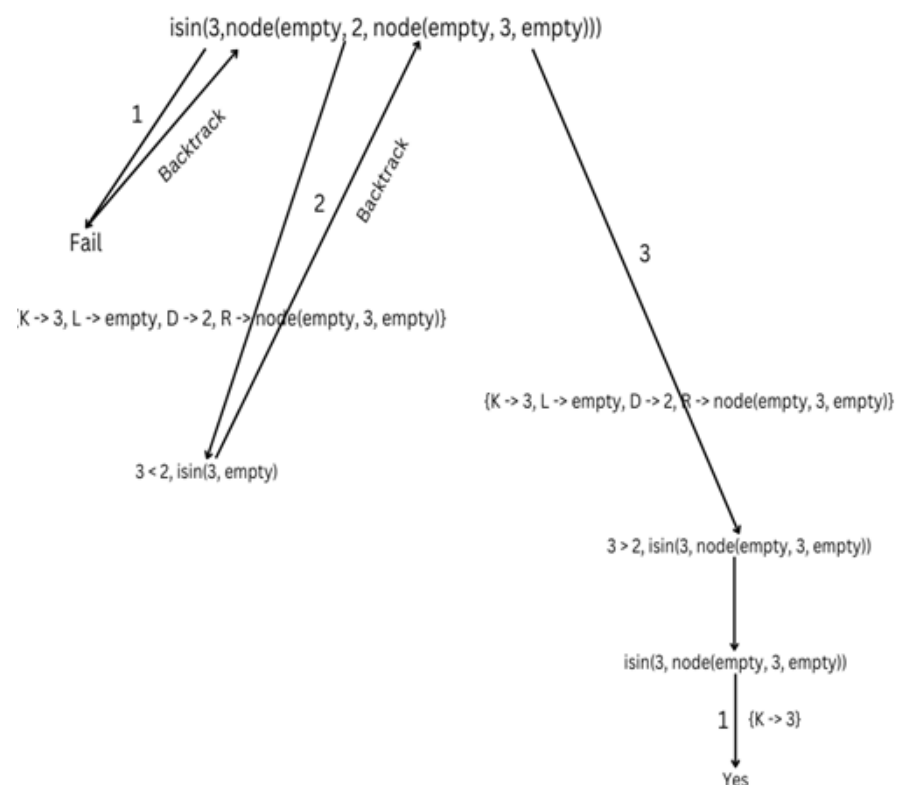
/ * 2 */ `isin(K, node(L, D, R)) :- K < D, isin(K, L)`.

/ * 3 */ `isin(K, node(L, D, R)) :- K > D, isin(K, R)`.

and query:

`isin(3, node(empty, 2, node(empty, 3, empty)))`.

The search tree (also called the resolution tree) for this query is:



Asking for another solution just causes backtracking

Cuts and Negation-A cut (written !) is a subgoal that always succeeds. As a side effect, it prevents backtracking past a certain point.

$P :- Q_1, Q_2, \dots, Q_m, !, Q_{m+2}, \dots, Q_n.$

After Q_m succeeds for the first time, no other possibilities (alternative rule applications for) P, Q_1, Q_2, \dots, Q_m are ever considered.

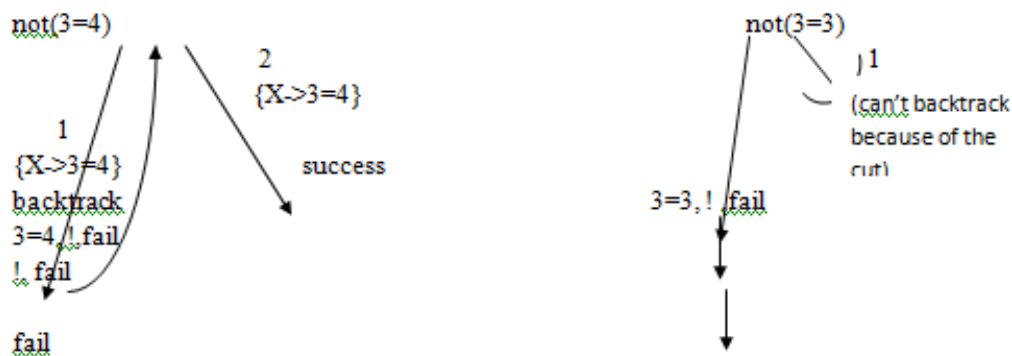
Cuts do not possess any logical interpretation -they are used just for efficiency. Cuts can be used to implement negations as shown below:

* 1 */ not(X):- X, !, fail.

* 2 */ not(X):- X,

Now, if X succeeds, then the definition of not fails because the cut keeps the second rule from being reached. If X fails, then the cut is never reached, so the interpreter backtracks to the second case and succeeds.

Examples:



Stop to Consider

A logic program is a kind of declarative program in that it describes the domain of the program and the goals the programmer would like to achieve. It emphasizes on what is true and what is wanted rather than how to achieve the desired goals.

CHECK YOUR PROGRESS

1. Assign a truth value for each of the following:

- i) $5 < 5 \vee 5 < 6$
- ii) $5 \times 4 = 21 \vee 9 + 7 = 17$
- iii) $6 + 4 = 10 \vee 0 > 2$
- iv) $(\forall x \in \mathbb{Z}) x^2 = x$
- v) $(\exists x \in \mathbb{Z}) x^2 = x$

2. Fill in the blanks with correct answer:

- i) A _____ is a function that returns Boolean value also called a predicate.
- ii) A fact is a rule with no _____.
- iii) A _____ is a question about a relation that is given to the Prolog interpreter.
- iv) _____ occurs when the current choice of a rule fails.
- v) A _____ is a subgoal that always succeeds.

3.7 SUMMING UP

Logic programming can significantly improve data-driven architectures by providing a declarative and knowledge-driven approach to data processing and analysis. It has ability to represent complex relationships and perform logical reasoning that can help obtaining meaningful information from vast amounts of data.

3.8 ANSWERS TO CHECK YOUR PROGRESS

- 1. i) **True** ii) **False** iii) **True** iv) **False** v) **True**
- 2. i) **relation** ii) **R.H.S** iii) **query** iv) **Backtracking** v) **cut (!)**

3.9 POSSIBLE QUESTIONS

- 1. Why Logic is considered as language for problem solving. Explain.
- 2. Discuss the basic terms used in Logic programming.

3. Describe the general structure and computational behavior of logic program.
4. What is Backtracking. Give example.
5. State the use of cut in logic programming.

3.10 REFERENCES AND SUGGESTED READINGS

- *Programming in Prolog: Using TheIso Standard* William F. Clocksin.
- *The Art of Prolog: Advanced Programming Techniques* Leon Sterling.
- *Adventure in Prolog* Dennis Merritt.

---X---

UNIT 4: LOGIC PROGRAMMING LANGUAGES - II

Unit Structure:

- 4.1 Introduction
- 4.2 Unit Objectives
- 4.3 Unification algorithm
 - 4.3.1 Conditions for unification
 - 4.3.2 Unify Algorithm
- 4.4 Procedural interpretation of Logic
- 4.5 Algorithmic view of logic program execution
- 4.6 A brief introduction to PROLOG
 - 4.6.1 Installing SWI Prolog on Windows
 - 4.6.2 Prolog Syntax and Programming
- 4.7 Summing Up
- 4.8 Answers to Check Your Progress
- 4.9 Possible Questions
- 4.10 References and Suggested Readings

4.1 INTRODUCTION

Prolog is a logic programming language. It has significant role in artificial intelligence. Not like many other programming languages, intentionally Prolog is a declarative programming language. In prolog programming, logic is expressed as relations (called as Facts and Rules). The main part of a prolog program lies at the **logic** being applied. Formulation or Computation is carried out by executing a query over these relations.

4.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Understand unification process used in prolog program
- Explain procedural and logical interpretation of a prolog program
- Install and do programming in PROLOG

4.3 UNIFICATION ALGORITHM

When a rule is applied, the current subgoal is unified against the rule's goal. The result of unification is substitution (set of values for variables) that make the subgoal and the goal identical. For example:

isin(3, node(empty, 3, empty))unifies with:

isin(K, node(L, K, R))

producing the substitution $\{K \rightarrow 3, L \rightarrow \text{empty}, R \rightarrow \text{empty}\}$

Example:

mem(3, [1, 2, 3])

unifies with:

mem(X, [Y | Z])

producing the substitution $\{X \rightarrow 3, Y \rightarrow 1, Z \rightarrow [2, 3]\}$

In the presentation of the Prolog interpreter algorithm, “matches” really means “unifies with”.

When a rule is used, unification between the current subgoal in the subgoal list and the goal of the rule takes place, and the resulting substitution is applied to the rest of the subgoal list. Applying a substitution implies replacing the variables in the subgoals with the values for those variables specified in the substitution.

Stop to Consider

Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process. It takes two literals as input and makes them identical using substitution. Let Ψ_1 and Ψ_2 be two atomic sentences and σ be a unifier such that, $\Psi_1\sigma = \Psi_2\sigma$, then it can be expressed as **UNIFY (Ψ_1, Ψ_2).**Substitution $\theta = \{Anil/y\}$

is a unifier for these atoms and applying this substitution, and both expressions will be identical.

The UNIFY algorithm is used for unification, which takes two atomic sentences and returns a unifier for those sentences (If any exist).Unification is a key component of all first-order inference algorithms. It returns fail if the expressions do not match with each other.

4.3.1 Conditions for unification

Following are some basic conditions for unification:

- Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

4.3.2 UNIFY Algorithm

Step 1: If Ψ_1 or Ψ_2 is a variable or constant, then:

- a) If Ψ_1 or Ψ_2 are identical, then return NIL.
- b) Else if Ψ_1 is a variable,
 - a. Then if Ψ_1 occurs in Ψ_2 , then return FAILURE
 - b. Else return $\{(\Psi_2/\Psi_1)\}$.
- c) Else if Ψ_2 is a variable,
 - a. If Ψ_2 occurs in Ψ_1 then return FAILURE,
 - b. Else return $\{(\Psi_1/\Psi_2)\}$.
- d) Else return FAILURE.

Step 2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step 3: IF Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

Step 4: Set Substitution set (SUBST) to NIL.

Step 5: For $i=1$ to the number of elements in Ψ_1 .

- a) Call Unify function with the i th element of Ψ_1 and i th element of Ψ_2 , and put the result into S.
- b) If $S=\text{failure}$ then returns Failure
- c) If $S \neq \text{NIL}$ then do,
 - a. Apply S to the remainder of both L1 and L2.
 - b. $\text{SUBST}=\text{APPEND}(S, \text{SUBST})$.

Step.6: Return SUBST.

Prolog programs can be read as statements in logic, or as programs to be executed.

Stop to Consider

Prolog stands for programming in logic. In the logic programming paradigm, prolog language is most widely available. Prolog is a declarative language, which means that a program consists of data based on the facts and rules (Logical relationship) rather than computing how to find a solution. A logical relationship describes the relationships which hold for the given application. The first Prolog was 'Marseille Prolog', which is based on work by Colmerauer. The major example of fourth-generation programming language is prolog. It supports the declarative programming paradigm. In 1981, a Japanese computer Project of 5th generation was announced. After that, it was adopted- Prolog as a development language.

4.4 PROCEDURAL INTERPRETATION OF LOGIC

Prolog interpreter must decide:

- what order to consider the goals of a query in
- what order to consider rules in
- what order to consider subgoals of a rule in

Logically these decisions have no consequence, but practically ordering effects efficiency and also whether a solution is obtained or infinite “looping” occurs.

Prolog interpreters:

- consider goals in a query from left to right
- consider rules by order in the program (using the first rule that “matches” a goal)
- consider subgoals in a rule from left to right.

The Prolog interpreter algorithm:

To solve a query Q_1, Q_2, \dots, Q_n .

Start with subgoal list Q_1, Q_2, \dots, Q_n

While the subgoal list is not empty do choose the leftmost subgoal Q_i

If some rule's goal “matches” Q_i then choose the first rule $G:-K_1, K_2, \dots, K_m$ such that G “matches” Q_i

Replace Q_i by K_1, K_2, \dots, K_m in the subgoal list

Else if backtracking is possible then backtrack

```

Else
    fail endif
end loop
succeed

```

where fail implies that the interpreter answers no, and succeed implies that the interpreter answers yes.

Notes:

- the subgoal list only shrinks when facts are used. Hence, put facts before other rules with the same goal in programs.
- leftmost subgoals of rules are applied first. Hence, the “most restrictive”(quickest to fail) subgoals should always be put first. For example, in the rule:

Isin (K, node (L, D, R)):- K < D, isin (K, L).

K<D is the first subgoal because it is quick to evaluate, and there is no reason to execute is in (K, L) if K<D fails.

4.5 ALGORITHMIC VIEW OF LOGIC PROGRAM EXECUTION

As previously discussed, subgoals of a rule are implicitly conjoined, multiple rules with the same goal are disjoined, and the :- operator is a reverse implication, so that:

P:-Q1,Q2,...,Qn

P:- K1,K2,..., Km:

is equivalent to:

$(Q1 \wedge Q2 \wedge \dots \wedge Qn \Rightarrow P)$

$\forall (K1 \wedge K2 \wedge \dots \wedge Km \Rightarrow P)$

In a Prolog rule, all variables appearing in the goal are universally quantified. Variables that appear only in subgoals are existentially quantified."

Examples:

grandparent (A, B) :- parent (A, C), parent (C,B).

means : $\forall A,B \exists C$ such that $\text{parent}(A, C) \wedge \text{parent}(C, B) \Rightarrow \text{grandparent}(A,B)$

Stop to Consider

In prolog program, we usually declare some facts. These facts comprises the Knowledge Base of the system. We can query against the Knowledge Base. We get output/result as positive if our query is already in the knowledge Base or it is implied by Knowledge Base, or else we get output as negative. So, Knowledge Base can be considered similar to database against which we can query. Prolog facts are expressed in distinct pattern. Facts include entities and their relation.

4.6 A BRIEF INTRODUCTION TO PROLOG

Prolog is a logic-based programming language that is extensively used in artificial intelligence, natural language processing, and expert systems. Some other applications of PROLOG are:

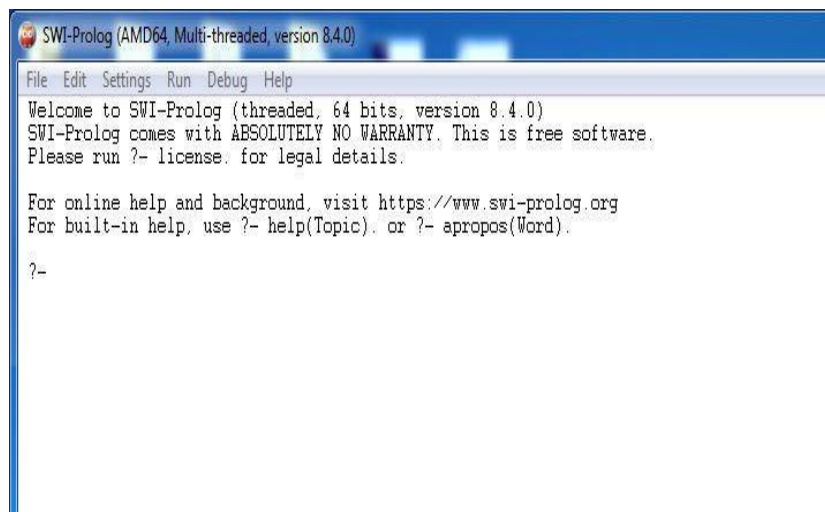
- Specification Language
- Robot Planning
- Machine Learning
- Problem Solving
- Intelligent Database retrieval
- Automated Reasoning

4.6.1 Installing SWI Prolog on Windows:

Below are the steps in brief to install SWI Prolog on Windows:

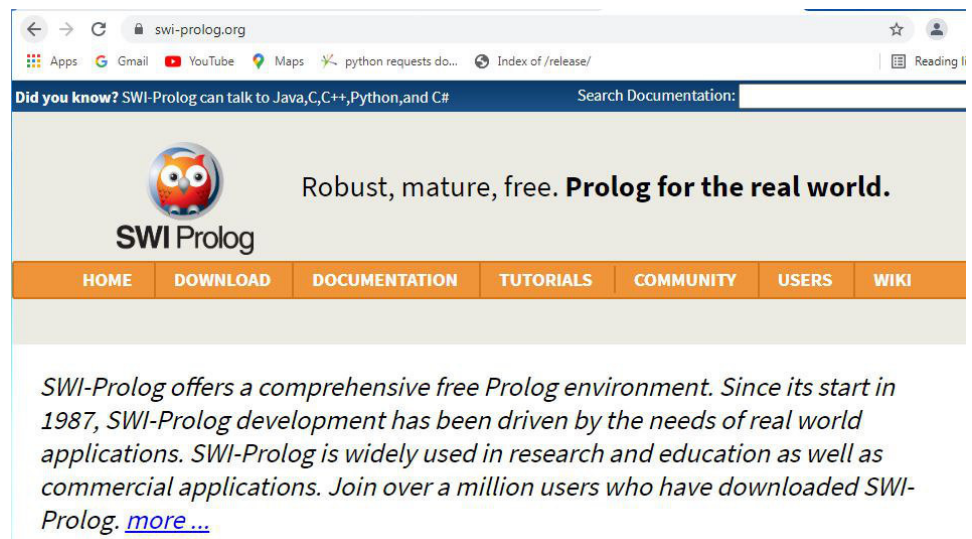
- **Step1:** Visit swi-prolog.org website using any web browser.
- **Step2:** Click on Download which is adjacent to Home, dropdown list will appear then click on SWI-Prolog.
- **Step3:** New webpage will open, click on Stable release.
- **Step4:** Download binaries
- **Step5:** Install binaries

Interface to write Prolog code is:

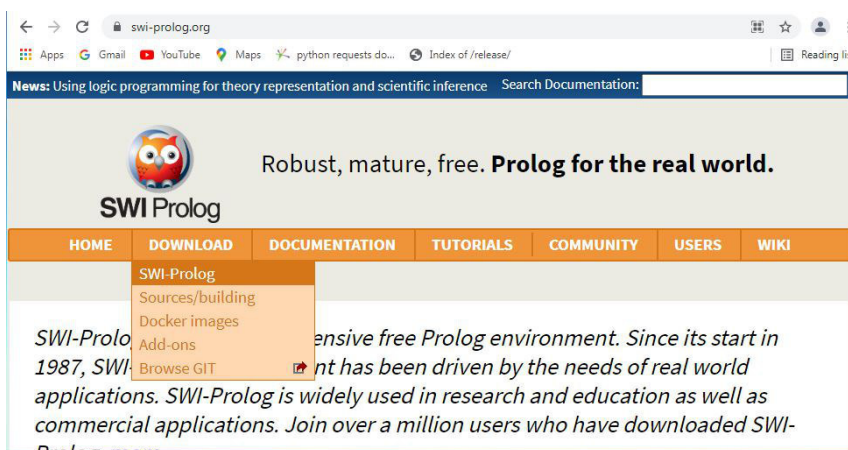


Follow the below steps to install SWI Prolog on Windows:

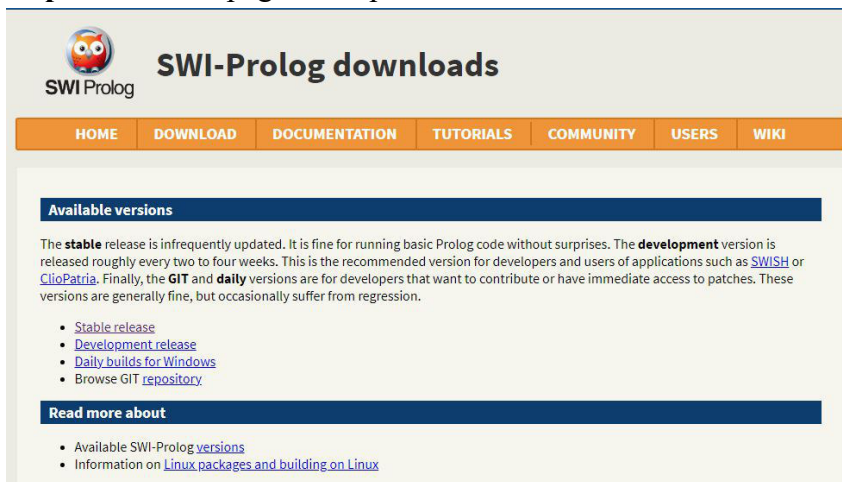
Step 1: Visit *swi-prolog.org* website using any web browser.



Step 2: Click on Download which is adjacent to Home, dropdown list will appear then click on SWI-Prolog.




Step 3: New webpage will open, click on Stable release.



Step 4: After clicking on stable release new webpage will open which will contain stable versions of prolog for different platforms. Under binaries there are two stable releases for windows, first is SWI-Prolog 8.4.0-1 for Microsoft Windows (64 bit) and the other is SWI-Prolog 8.4.0-1 for Microsoft Windows (32 bit). Click on the one as per your system configuration. Lets take the one for 64-bit operating system.

News: Using logic programming for theory representation and scientific inference
Search Documentation:



Download SWI-Prolog stable versions



HOME
DOWNLOAD
DOCUMENTATION
TUTORIALS
COMMUNITY
USERS
WIKI

Linux versions are often available as a package for your distribution. We collect information about available packages and issues for building on specific distros [here](#). We provide a [PPA](#) for [Ubuntu](#) and [snap images](#)

Android binaries are available for [Termux](#) as the package `swi-prolog`. See also [Building SWI-Prolog on Android using LinuxOnAndroid](#)


Please check the [windows release notes](#) (also in the SWI-Prolog startup menu of your installed version) for details.

⚠️ Examine the [ChangeLog](#).

Binaries		
	12,469,294 bytes	SWI-Prolog 8.4.0-1 for Microsoft Windows (64 bit) Self-installing executable for Microsoft's Windows 64-bit editions. Requires at least Windows 7. See the reference manual for deciding on whether to use the 32- or 64-bits version. This binary is linked against GMP 6.1.1 which is covered by the LGPL license. SHA256: 787f8de518dba327a6776e755478e49ee39917c443b4787b38f7fe44e6b5cecb
	12,452,305 bytes	SWI-Prolog 8.4.0-1 for Microsoft Windows (32 bit) Self-installing executable for MS-Windows. Requires at least Windows 7. Installs swipl-win.exe and swipl.exe . This binary is linked against GMP 6.1.1 which is covered by the LGPL license.

Step 5. After clicking on SWI-Prolog 8.4.0-1 for Microsoft Windows (64 bit), a new webpage will open, check on I understand checkbox to make the download link active. Then click on the download link, downloading of the executable file will start shortly. It is a small 11.9 MB file that will hardly take a minute.

News: Using logic programming for theory representation and scientific inference
Search Documentation:



Download binary

HOME
DOWNLOAD
DOCUMENTATION
TUTORIALS
COMMUNITY
USERS
WIKI

⚠️ Windows antivirus software works using *signatures* and *heuristics*. Using the huge amount of viruses and malware known today, arbitrary executables are often [falsely classified as malicious](#). [Google Safe Browsing](#), used by most modern browsers, therefore often classifies our Windows binaries as malware. You can use e.g., [virustotal](#) to verify files with a large number of antivirus programs.

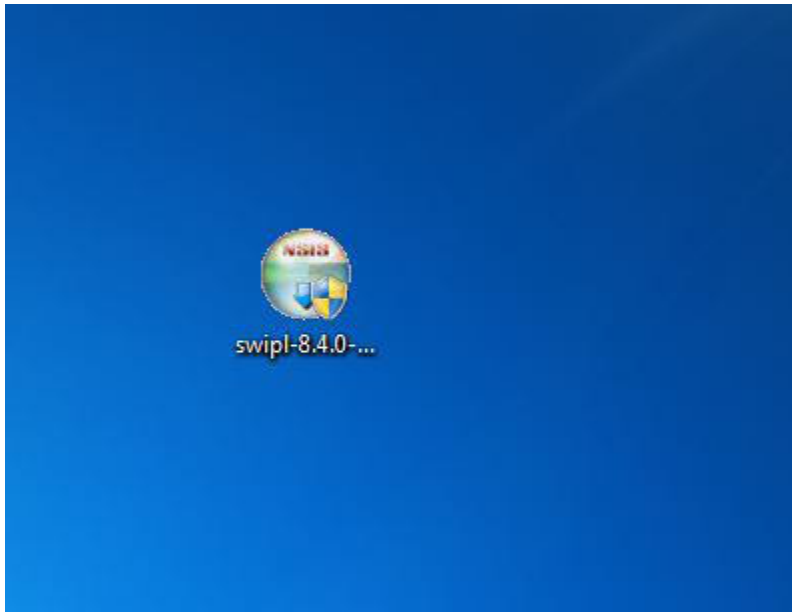
Our Windows binaries are cross-compiled on an isolated Linux container. The integrity of the binaries on the server is regularly verified by validating its SHA256 fingerprint.

Please select the checkbox below to enable the actual download link.

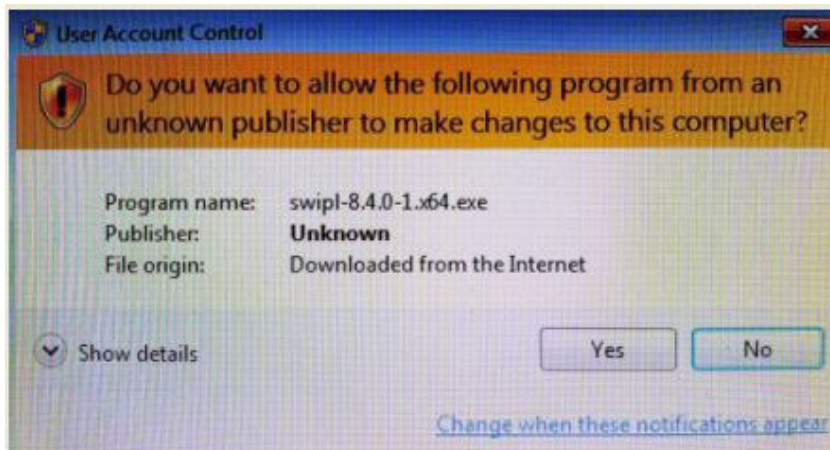
☒ I understand

[Download swipl-8.4.0-1.x64.exe](#) (SHA256: 787f8de518dba327a6776e755478e49ee39917c443b4787b38f7fe44e6b5cecb)
[VIRUSTOTAL Scan Result](#)

Step 6: Now check for the executable file in downloads in your system and run it.



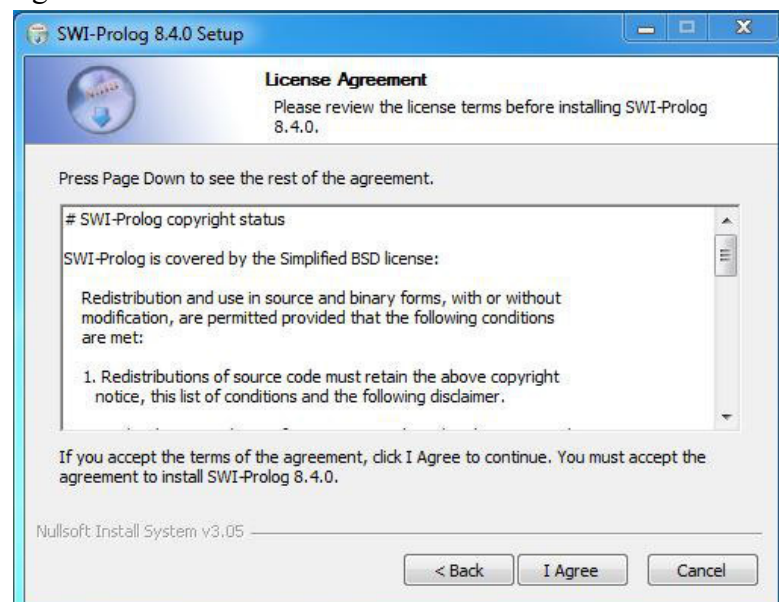
Step 7: It will prompt confirmation to make changes to your system. Click on Yes.



Step 8: Setup screen will appear, click on Next.

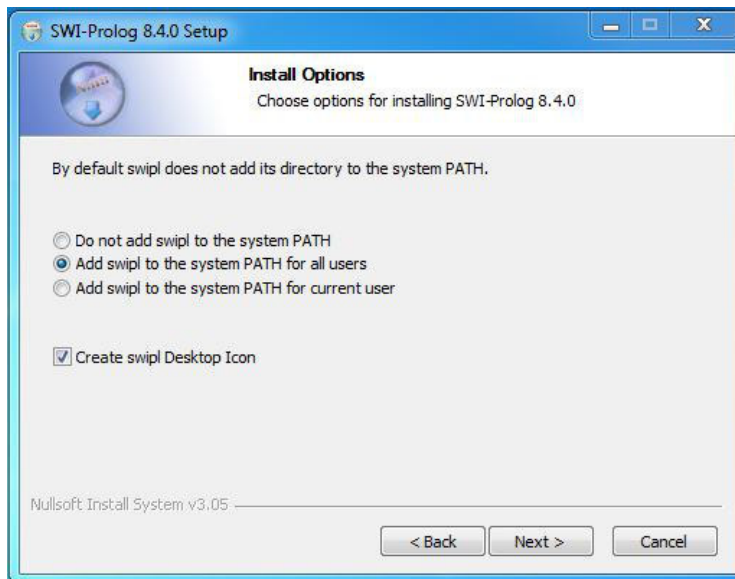


Step 9: The next screen will be of License Agreement, click on I Agree.



Step 10: After it there will be screen of installing options so check the box for Add swipl to the system path for all users, and also

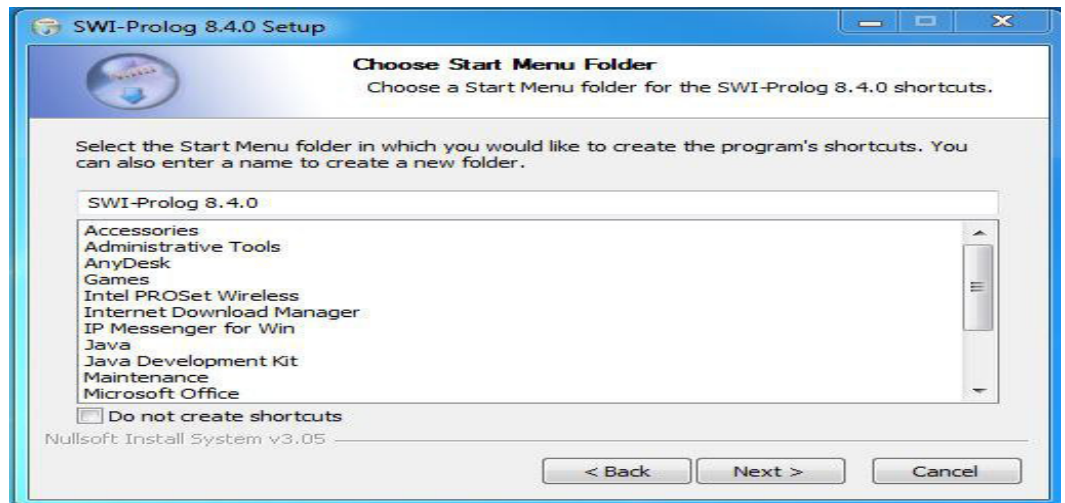
check the box for create a desktop icon and then click on the Next button.



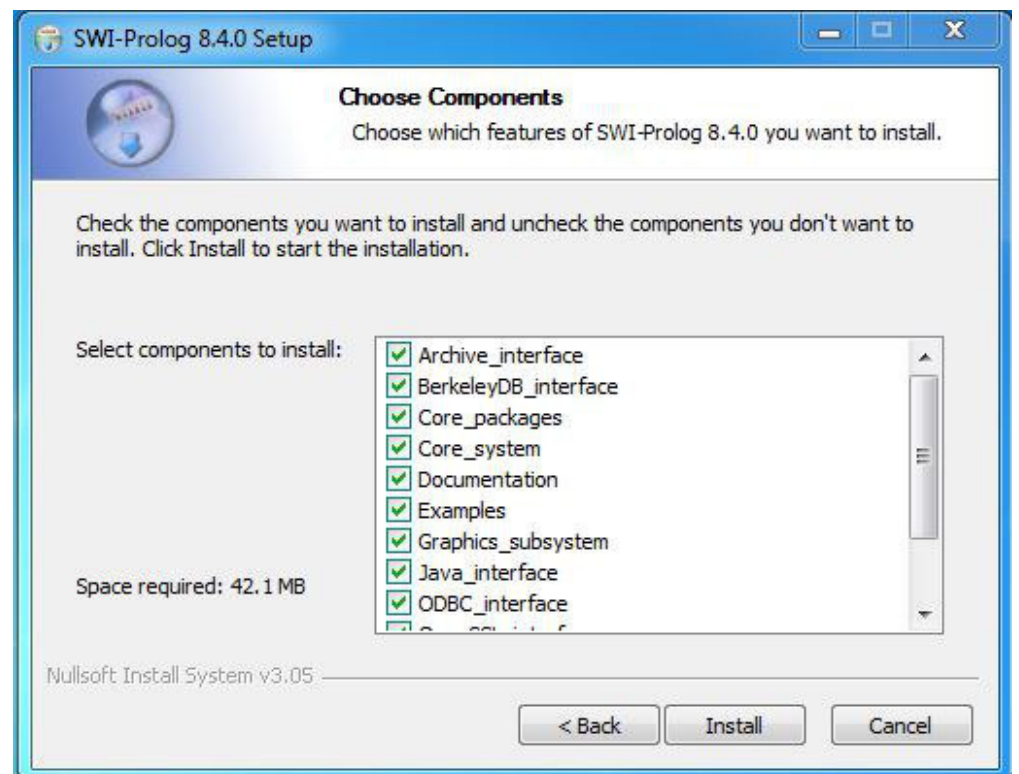
Step 11: The next screen will be of installing location so choose the drive which will have sufficient memory space for installation. It needed only a memory space of 50 MB.



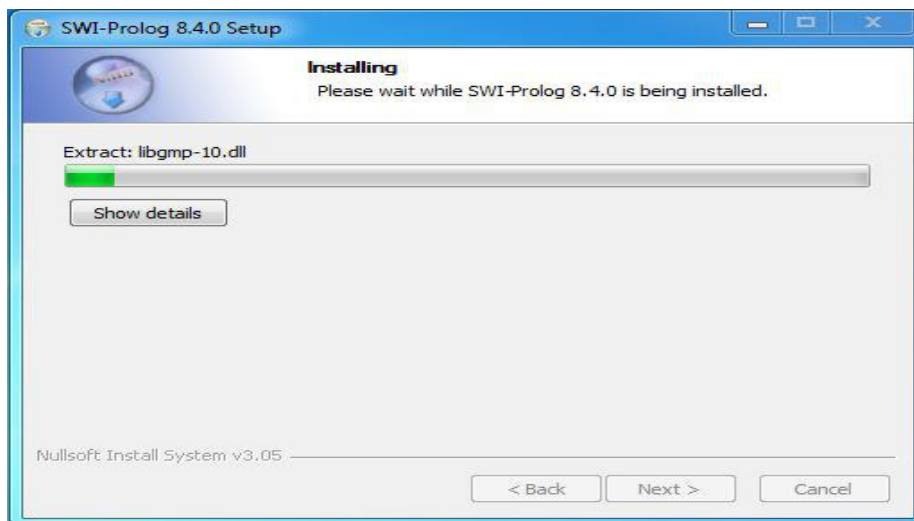
Step 12: Next screen will be of choosing Start menu folder so don't do anything just click on Next Button.



Step 13: This last screen is of choosing components, all components are already marked so don't change anything just click on Install button.



Step 14: After this installation process will start and will hardly take a minute to complete the installation.



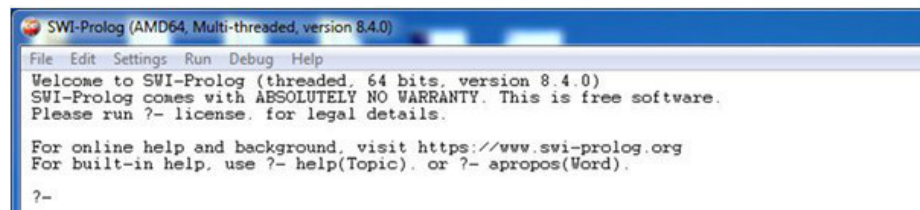
Step 15: Click on Finish after the installation process is complete



Step 16: SWI Prolog is successfully installed on the system and an icon is created on the desktop.



Step 17: Run the software and see the interface.



How to run a SWI-PROLOG program: 1. Open the SWI-Prolog application installed on your system. A GUI as shown below will appear:



2. Select of the “File” option and click “new”
3. A new GUI will appear, write a simple Swi-prolog program and save (ctrl + s) the file with .pl extension.

```

pqr.pl
File Edit Browse Compile Prolog Pcs Help
xyz.pl pqr.pl
%Facts
parent(john,mary).
parent(mary,alice).
%rule
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).

```

4. Now to run the swi-prolog, go back to the interface as shown in step-1. Under the “file” menu click on “consult” and select the saved file and run the program.

```

SWI-Prolog (AMD64, Multi-threaded, version 9.2.9)
File Edit Settings Run Debug Help
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
ERROR: [Thread pce] c:/users/achar/onedrive/documents/prolog/pqr.pl:7:46: Syntax
error: Unexpected end of file
% c:/Users/achar/OneDrive/Documents/Prolog/pqr.pl compiled 0.03 sec, 1 clauses
ERROR: [Thread pce] c:/users/achar/onedrive/documents/prolog/pqr.pl:7:45: Syntax
error: Unexpected end of file
% c:/Users/achar/OneDrive/Documents/Prolog/pqr.pl compiled 0.02 sec, -1 clauses
Warning: c:/users/achar/onedrive/documents/prolog/pqr.pl:7:
Warning: Singleton variables: [X,Z]
% c:/Users/achar/OneDrive/Documents/Prolog/pqr.pl compiled 0.00 sec, 1 clauses
% c:/Users/achar/OneDrive/Documents/Prolog/pqr.pl compiled 0.00 sec, 0 clauses
?- grandparent(john,alice).
true.

?- parent(john,alice).
false.

?-

```

4.6.2 Prolog Syntax and Programming

Hello World Program

- After running the SWI prolog, we can write hello world program directly from the console.
- To do so, we have to write the command as follows:

write('Hello World').

Note: After each line, you have to use one period (.) symbol to show that the line has ended.

Write codes in Files

- Now create one file (test.pl) and write the code as follows:

**main :-write('This is sample Prolog program'),
write(' This program is written into test.pl file').**

- Now let's run the code.
- To run it, we have to write the file name as follows: [test].

- After that, type

man.

Define facts in Prolog

- We can define fact as an explicit relationship between objects, and properties these objects might have.
- So facts are unconditionally true in nature.
- Suppose we have some facts as given below:

Facts	Prolog command
Tom is a cat	cat (tom).
Jahir loves to eat Pasta	love_to_eat (jahir,pasta).
Hair is black	of_color (hair,black).
Mahmuda loves to play games	love_to_play_games (mahmuda).
Jahangir is lazy.	Lazy (Jahangir).

Rules

- We can define rule as an implicit relationship between objects.
- So facts are conditionally true.
- So when one associated condition is true, then the predicate is also true.
- Suppose we have some rules as given below:

Rules	Prologcommand
Lili is happy if she dances.	happy(lili):-dances(lili).
Tom is hungry if he is searching for food.	hungry(tom):- search_for_food(tom).
Jack and Bili are friends if both of them love to play cricket.	friends(jack,bili):- lovesCricket(jack),lovesCricket(bili).
Ryan will go to play if school is closed, and he is free.	goToPlay(ryan):- isClosed(school),free(ryan).

Suppose a clause is like : P :-Q;R.

- This can also be written as

P :-Q.

P :-R.

- Suppose a clause is like : P :- (Q,S);(R,T).

- This can also be written as

P :-Q,S.

P :-R,T.

Queries

- Queries are some questions on the relationships between objects and object properties.

- So question can be anything, as given below:

- So according to these queries, Logic programming language can find the answer and return them.

Rules

Rules	Prologcommand
Is tom a cat?	cat(tom).
Does Jahir love to eat pasta?	loves_to_eat(jahir,pasta).
Is Lili happy?	happy(lili).
Will Ryan go to play?	go_to_play(ryan).

Lets implement in Prolog

- Knowledge base

sing_a_song(ananya).

listens_to_music(rohit).

listens_to_music(ananya) :-sing_a_song(ananya).

happy(ananya) :-sing_a_song(ananya).

happy(rohit) :-listens_to_music(rohit).

plays_guitar(rohit) :-listens_to_music(rohit).

Get answer

- Suppose we want to see the members who plays guitar, we can use one variable in our query.

- **The variables** should start with uppercase letters.

plays_guitar(X).

Relations

- In Prolog programs, it specifies relationship between objects and properties of the objects.
- Suppose, there's a statement,
- “Amit has a bike”, then we are actually declaring the ownership relationship between two objects —
- one is Amit and the other is bike.
- If we ask a question, “Does Amit own a bike?”, we are actually trying to find out about one relationship.
- There are various kinds of relationships, of which some can be rules as well.
- A rule can find out about a relationship even if the relationship is not defined explicitly as a fact.

We can define a brother relationship as follows:

Two person are brothers, if,

- They both are male.
- They have the same parent.

Now consider we have the below phrases:

- parent(sudip, piyus).
- parent(sudip, raj).
- male(piyus).
- male(raj).
- brother(X,Y) :-parent(Z,X), parent(Z,Y),male(X), male(Y),X\==Y.

Try this in Prolog

- female(pam).
- female(liz).
- female(pat).
- female(ann).
- male(jim).
- male(bob).
- male(tom).
- male(peter).

- parent(pam,bob).
- parent(tom,bob).
- parent(tom,liz).
- parent(bob,ann).
- parent(bob,pat).
- parent(pat,jim).
- parent(bob,peter).
- parent(peter,jim).

- mother(X,Y):-parent(X,Y),female(X).
- father(X,Y):-parent(X,Y),male(X).
- sister(X,Y):parent(Z,X),parent(Z,Y),female(X),X\==Y.
- brother(X,Y):-parent(Z,X),parent(Z,Y),male(X),X\==Y.
- grandparent(X,Y):-parent(X,Z),parent(Z,Y).
- grandmother(X,Z):-mother(X,Y),parent(Y,Z).
- grandfather(X,Z):-father(X,Y),parent(Y,Z).
- wife(X,Y):-parent(X,Z),parent(Y,Z),female(X),male(Y).
- uncle(X,Z):-brother(X,Y),parent(Y,Z).

Strings of characters enclosed in single quotes.

- This is useful if we want to have an atom that starts with a capital letter.
- By enclosing it in quotes, we make it distinguishable from variables:
- ‘Ramiz’
- ‘Majharul’
- ‘Kamrun’

Anonymous Variables in Prolog

- Anonymous variables have no names.
- The anonymous variables in prolog is written by a single underscore character ‘_’.
- And one important thing is that each individual anonymous variable is treated as different.
- They are not same.
- Now the question is, where should we use these anonymous variables?
- Suppose in our knowledge base we have some facts —“jim hates tom”, “pat hates bob”.

- So if tom wants to find out who hates him, then he can use variables.
- However, if he wants to check whether there is someone who hates him, we can use anonymous variables.
- So when we want to use the variable, but do not want to reveal the value of the variable, then we can use anonymous variables.

Comparison Operators

Comparison operators are used to compare two equations or states. Following are different comparison operators:

Operator	Meaning
$X > Y$	X is greater than Y
$X < Y$	X is less than Y
$X \geq Y$	X is greater than or equal to Y
$X \leq Y$	X is less than or equal to Y
$X == Y$	the X and Y values are equal
$X \neq Y$	the X and Y values are not equal

Arithmetic Operator

Arithmetic operators are used to perform arithmetic operations. There are few different types of arithmetic operators as follows:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power

//	Integerdivision
mod	Modulus

Example program

```

calc :-X is 100 + 200, write('100 + 200 is '),write(X),nl,
Y is 400 -150,write('400 -150 is '),write(Y),nl,
Z is 10 * 300,write('10 * 300 is '),write(Z),nl,
A is 100 / 30,write('100 / 30 is '),write(A),nl,
B is 100 // 30,write('100 // 30 is '),write(B),nl,
C is 100 ** 2,write('100 ** 2 is '),write(C),nl,
    D is 100 mod 30,write('100 mod 30 is '),write(D),nl.

```

Loops in Prolog

- Loop statements are used to execute the code block multiple times.
- In general, for, while, do-while are loop constructs in programming languages (like Java, C, C++).
- Code block is executed multiple times using recursive predicate logic.
- There are no direct loops in some other languages, but we can simulate loops with few different techniques.

Example

```

count_to_10(10) :-write(10),nl.
count_to_10(X) :-
write(X),nl,
Y is X + 1,
    count_to_10(Y).

```

Loop

- Now create a loop that takes lowest and highest values.
- So, we can use the between () to simulate loops.
- Example:

```

count_down(H,L) :-
between(L, H, Y),
Z is H -Y+1,
write(Z), nl.
count_up(L, H) :-
between(L, H, Y),
    write(Y), nl.

```

Decision Statements

- The decision statements are If-Then-Else statements.
- So when we try to match some condition, and perform some task, then we use the decision making statements.
- The basic usage is as follows:
- If is true, Then , Else
- Example:

```
gt(X,Y):-X >= Y,write(X),write(' is greater or equal'), write(Y).  
gt(X,Y):-X < Y,write(Y),write(' is greater'), write(X).
```

Conjunction & Disjunction

- Conjunction (AND logic) can be implemented using the comma (,) operator.
- Disjunction (OR logic) can be implemented using the semi-colon (;) operator.

Lists in Prolog

- The list is a simple data structure that is widely used in non-numeric programming.
- List consists of any number of items, for example, red, green, blue, white, dark.
- It will be represented as, [red, green, blue, white, dark].
- The list of elements will be enclosed with square brackets.
- A list can be either empty or non-empty.
- In the first case, the list is simply written as a **Prolog atom, []**.
- In the second case, the list consists of two things as given below:
- The first item, called the **head** of the list;
- The remaining part of the list, called the **tail**.

Now, let us consider we have a list, $L = [a, b, c]$.

- If we write $Tail = [b, c]$ then we can also write the list L as $L = [a | Tail]$.
- Here the vertical bar (|) separates the head and tail parts.

Operations on list

Opeartion	Defintion
Membership Checking	During this operation, we can verify whether a given element is member of specified list or not?
Length Calculation	With this operation, we can find the length of a list.
Concatenation	Concatenation is an operation which is used to join/add two lists.
Delete Items	This operation removes the specified element from a list.
Append Items	Append operation adds one list into another (as an item).
Insert Items	This operation inserts a given item into a list.

List Member

- To design this predicate, we can follow these observations.
- X is a member of L if either:
- X is head of L, or
- X is a member of the tail of L
- Program

`list_member(X,[X|_]).`

`list_member(X,[_|TAIL]) :-list_member(X,TAIL).`

Length Calculation

- This is used to find the length of list L.
- We will define one predicate to do this task.
- Suppose the predicate name is `list_length(L,N)`.
- This takes L and N as input argument.

- This will count the elements in a list L and instantiate N to their number.
- As was the case with our previous relations involving lists, it is useful to consider two cases:
- If list is empty, then length is 0.
- If the list is not empty, then $L = [\text{Head}|\text{Tail}]$, then its length is 1 + length of Tail.
- Program
- `list_length([],0).`
- `list_length([_|TAIL],N) :-list_length(TAIL,N1), N is N1 + 1.`

Concatenation:

- Concatenation of two lists means adding the list items of the second list after the first one.
- So if two lists are [a,b,c] and [1,2], then the final list will be [a,b,c,1,2].
- So to do this task we will create one predicate called `list_concat()`, that will take first list L1, second list L2, and the L3 as resultant list.
- There are two observations here.
- If the first list is empty, and second list is L, then the resultant list will be L.
- If the first list is not empty, then write this as [Head|Tail], concatenate Tail with L2 recursively, and store into new list in the form, [Head|NewList].
- Program

```
list_concat([],L,L).
```

```
list_concat([X1|L1],L2,[X1|L3]) :-list_concat(L1,L2,L3).
```

Delete from List

- Suppose we have a list L and an element X, we have to delete X from L.
- So there are three cases:
- If X is the only element, then after deleting it, it will return empty list.
- If X is head of L, the resultant list will be the Tail part.
- If X is present in the Tail part, then delete from there recursively.
- Program

```
list_delete(X, [X], []).
```

```
list_delete(X,[X|L1], L1).
```

```
list_delete(X, [Y|L2], [Y|L1]) :-list_delete(X,L2,L1).
```

Append into List

- Appending two lists means adding two lists together, or adding one list as an item.
- Now if the item is present in the list, then the append function will not work.
- So we will create one predicate namely, `list_append(L1, L2, L3)`.
- The following are some observations:
- Let A is an element, L1 is a list, the output will be L1 also, when L1 has A already.
- Otherwise new list will be $L2 = [A|L1]$.
- Program

```
list_member(X,[X|_]).  
list_member(X,[_|TAIL]) :-list_member(X,TAIL).  
list_append(A,T,T) :-list_member(A,T),!.  
list_append(A,T,[A|T]).
```

Insert into List:

- This method is used to insert an item X into list L, and the resultant list will be R.
- So the predicate will be in this form `list_insert(X, L, R)`.
- So this can insert X into L in all possible positions.
- If we see closer, then there are some observations.
- If we perform `list_insert(X,L,R)`, we can use `list_delete(X,R,L)`, so delete X from R and make new list L.
- Program

```
list_delete(X, [X], []).  
list_delete(X,[X|L1], L1).  
list_delete(X, [Y|L2], [Y|L1]) :-list_delete(X,L2,L1).  
list_insert(X,L,R) :-list_delete(X,R,L).
```

Experiment 1

- Write a prolog program to calculate the sum of two numbers.

Program:

```
sum(X,Y):-  
S is X+Y,  
format('The sum is '),  
write(S).
```

Experiment 2

- Write a prolog program to find the maximum of two numbers.

Program:

```
max(X,Y):-  
X=Y, write('both are equal').  
max(X,Y):-  
X>Y, Z is X, write('Maximum is '), write(Z),nl.  
max(X,Y):-  
    X<Y, Z is Y, write('Maximum is '), write(Z),nl.
```

Experiment 3

- Write a prolog program to calculate the factorial of a given number.

Program:

```
factorial(0,1).  
factorial(N,F):-  
N>0,  
N1 is N-1,  
factorial(N1,F1),  
    F is N*F1.
```

Experiment 4

- Write a prolog program to calculate the nth Fibonacci number.

Program:

```
fibonacci(S,N,F):-  
F>0,  
Y is S+N,  
write(S),write(', '),  
F1 is F-1,  
    fibonacci(N,Y,F1).
```

Experiment 5

- Write a prolog program, insert_nth(item, n, into_list, result) that asserts that result is the list into_list with item inserted as the nth element into every list at all levels.

Program:

```
insert_nth(L,1,Y,[L|Y]).  
insert_nth(L,P,[X|Y],[X|N]):-
```

```

P>0,
P1 is P -1,
    insert_nth(L,P1,Y,N).

```

Experiment 6

- Write a Prolog program to remove the nth item from a list

Program:

```

delete(1,[H|T],T).
delete(N,[H|T],[H|Y]):-
N>0,
N1 is N-1,
    delete(N1,T,Y).

```

Experiment 7

- Write a Prolog program to implement append for two lists

Program:

```

append([],L,L).
append([X|M],N,[X|Q]):-
    append(M,N,Q).

```

Experiment 8

- Write a Prolog program to implement palindrome (List).

Program:

```

append([],L,L).
append([X|M],N,[X|Q]):-
append(M,N,Q).
palind([]):-write('palindrome').
palind([_]):-write('palindrome').
palind(L) :-
append([H|T], [H], L),
palind(T)
;
    write('Not a palindrome').

```

Experiment 9

- Write a Prolog program to implement max(X,Y,Max) so that Max is the greater of two numbers X and Y.

Program:

max(A,B,Max):-

A>=B,

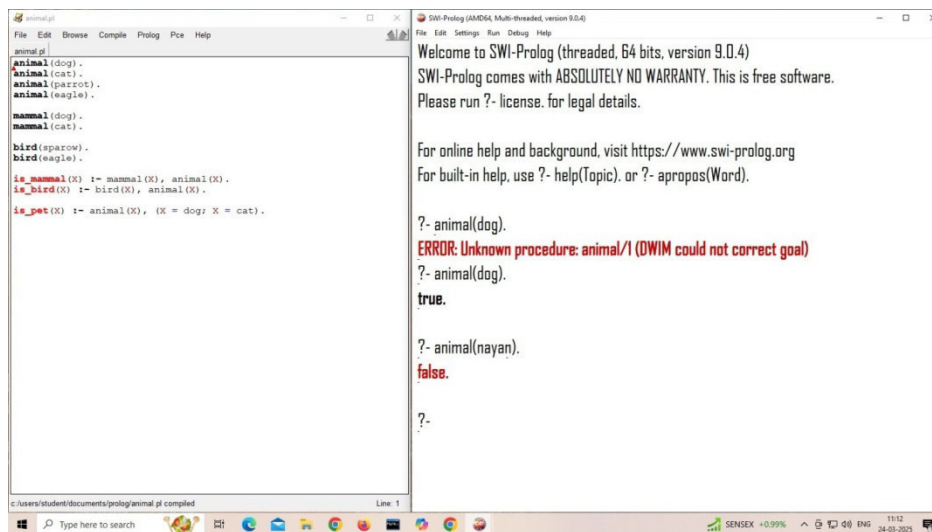
Max is A;

A<B,

Max is B.

Example Pprograms:

1. Program that check the given input is animal or not.



The screenshot shows the SWI-Prolog IDE with a Prolog program in the left pane and its execution results in the right pane.

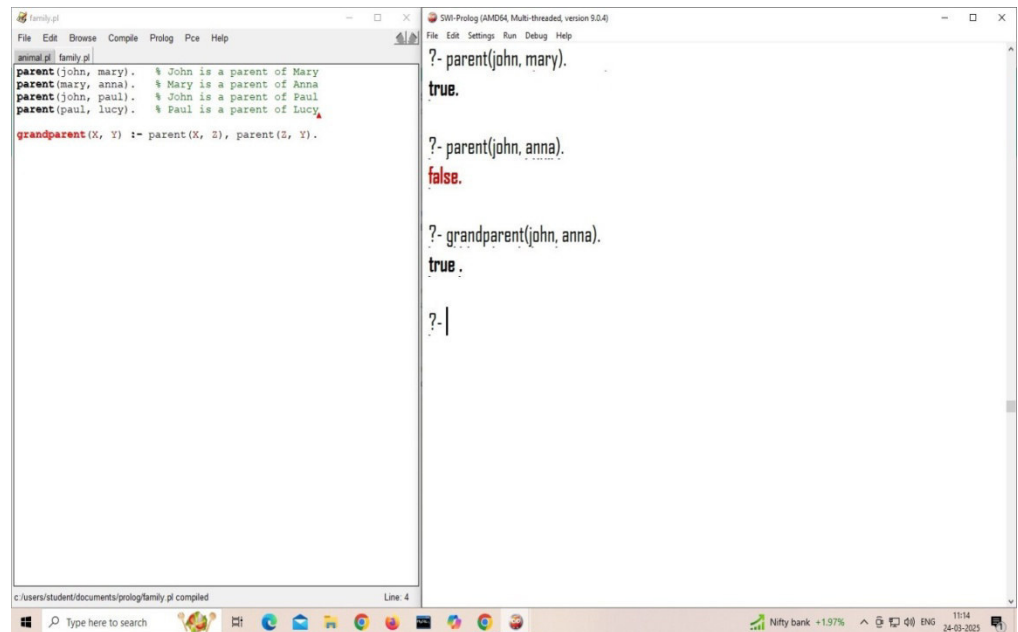
Program (Left Pane):

```
animal(dog).  
animal(cat).  
animal(parrot).  
animal(eagle).  
mammal(dog).  
mammal(cat).  
bird(sparrow).  
bird(eagle).  
is_mammal(X) :- mammal(X), animal(X).  
is_bird(X) :- bird(X), animal(X).  
is_pet(X) :- animal(X), (X = dog; X = cat).
```

Execution Results (Right Pane):

```
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.4)  
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.  
Please run ?- license. for legal details.  
  
For online help and background, visit https://www.swi-prolog.org  
For built-in help, use ?- help(Topic). or ?- apropos(Word).  
  
?- animal(dog).  
ERROR: Unknown procedure: animal/1 (DWIM could not correct goal)  
?- animal(dog).  
true.  
  
?- animal(nayan).  
false.  
  
?-
```


2. Program to check given input belongs to a family or not.



The screenshot shows the SWI-Prolog IDE with two windows. The left window, titled 'family.pl', contains the following Prolog code:

```
parent(john, mary). % John is a parent of Mary
parent(mary, anna). % Mary is a parent of Anna
parent(john, paul). % John is a parent of Paul
parent(paul, lucy). % Paul is a parent of Lucy

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

The right window shows the results of several queries:

```
?- parent(john, mary).
true.

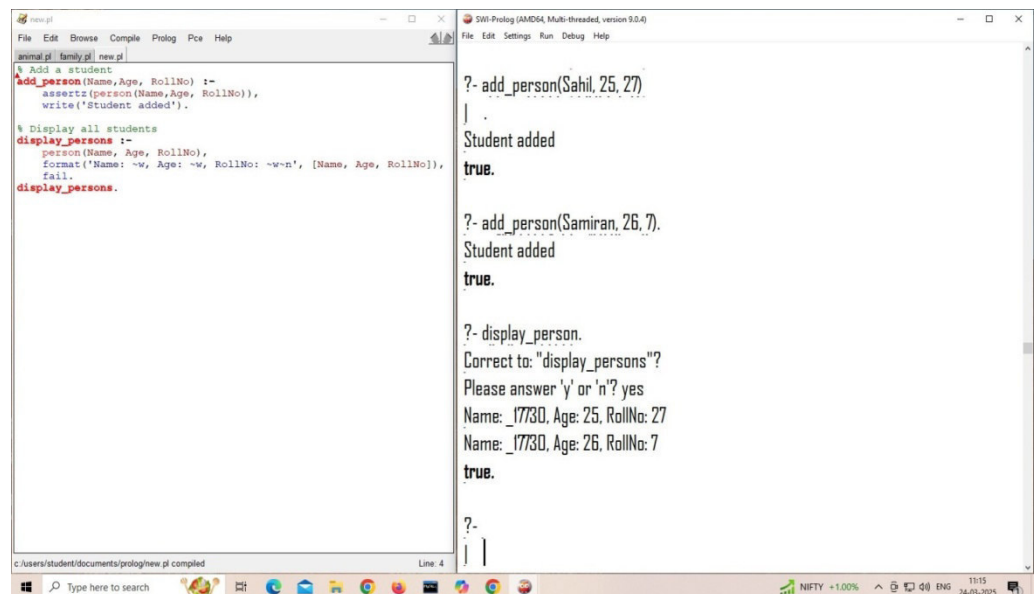
?- parent(john, anna).
false.

?- grandparent(john, anna).
true.

?- |
```

The status bar at the bottom indicates the file path 'c:/users/student/documents/prolog/family.pl compiled' and 'Line 4'.

3. Program to add student.



The screenshot shows the SWI-Prolog IDE with two windows. The left window, titled 'new.pl', contains the following Prolog code:

```
% Add a student
add_person(Name, Age, RollNo) :-
    assertz(person(Name, Age, RollNo)),
    write('Student added').

% Display all students
display_persons :-
    person(Name, Age, RollNo),
    format('Name: ~w, Age: ~w, RollNo: ~w~n', [Name, Age, RollNo]),
    fail.
display_persons.
```

The right window shows the results of several queries:

```
?- add_person(Sahil, 25, 27)
|
Student added
true.

?- add_person(Samiran, 26, 7).
Student added
true.

?- display_person.
Correct to: "display_persons"?
Please answer 'y' or 'n'? yes
Name: _17730, Age: 25, RollNo: 27
Name: _17730, Age: 26, RollNo: 7
true.

?- |
```

The status bar at the bottom indicates the file path 'c:/users/student/documents/prolog/new.pl compiled' and 'Line 4'.

Check Your Progress

1. State True or False for the following statements:

- i) The result of unification is substitution that does not make the subgoal and the goal identical.
- ii) The subgoal list only shrinks when facts are used.
- iii) Rightmost subgoals of rules are applied first.
- iv) In a Prolog rule, all variables appearing in the goal are existentially quantified.
- v) Variables in a Prolog rule, that appear only in subgoals are universally quantified.

4.7 SUMMING UP

PROLOG is a declarative programming language. That means it allows the programmer to specify the rules and facts about a problem domain, and after that Prolog interpreter will use these rules and facts to automatically infer solutions to problems. Prolog is deeply used in artificial intelligence, symbolic computation, and natural language processing.

4.8 ANSWERS TO CHECK YOUR PROGRESS

1. i) False ii) True iii) False iv) False v) False

4.9 POSSIBLE QUESTIONS

- 1. Discuss unification in Logic programming. Give examples.
- 2. State i) Procedural and ii) Logical interpretation of a PROLOG program.
- 3. Briefly explain PROLOG program execution.

4.10 REFERENCES AND SUGGESTED READINGS

- The Art of Prolog: Advanced Programming Techniques 2nd Printing Edition by Leon Sterling
- Other resources online: [Learn Prolog Now!](#)
- [swi-prolog](#) is an popular, high quality and well supported open source Prolog implementation. There are tutorials on the site.

---X---