Master of Computer Applications (MCA)

Object Oriented Programming with C++ and JAVA (DMCACO101T24)

Self-Learning Material (SEM 1)

Jaipur National University
Centre for Distance and Online Education

Established by Government of Rajasthan
Approved by UGC under Sec 2(f) of UGC ACT 1956
&
NAAC A+ Accredited

TABLE OF CONTENTS

EXPERT COMMITTEE

Prof. Sunil Gupta
(Department of Computer and Systems Sciences, JNU Jaipur)

Dr. Deepak Shekhawat
(Department of Computer and Systems Sciences, JNU Jaipur)

Dr. Shalini Rajawat
(Department of Computer and Systems Sciences, JNU Jaipur)

COURSE COORDINATOR

Ms. Heena Shrimali
(Department of Computer and Systems Sciences, JNU Jaipur)

UNIT PREPARATION

Unit Writer(s)	**Assisting & Proofreading**	**Unit Editor**
Ms. Heena Shrimali (Department of Computer and Systems Sciences, JNU Jaipur) (Unit 1 -5)	Mrs. Swarnima Gupta (Department of Computer and Systems Sciences, JNU Jaipur)	Mr. Ramlal Yadav (Department of Computer and Systems Sciences, JNU Jaipur)
Mr. Satender Singh (Department of Computer and Systems Sciences, JNU Jaipur) (Unit 6- 9)		
Ms. Rachna Yadav (Department of Computer and Systems Sciences, JNU Jaipur) (Unit 10-12)		

Secretarial Assistance

Mr. Mukesh Sharma

COURSE INTRODUCTION

"Object-oriented programming is an exceptionally bad idea which could only have originated in California."

- Edsger W. Dijkstra

Object-oriented programming (OOP) is a paradigm that has transformed software development by emphasizing objects and classes to create scalable and maintainable applications. This course explores key OOP principles and their implementation in both C++ and Java, providing students with a thorough understanding of modern programming practices. The main goals of this course are to introduce students to core OOP concepts such as classes, objects, inheritance, polymorphism, and encapsulation. Students will build a strong foundation in C++ and Java, utilizing the strengths of each language for various programming tasks. Through practical exercises and projects, students will gain experience in file handling, exception management, multithreading, and database connectivity using JDBC.

This course has 3 credits and is divided into 9 Units. The course starts with an overview of the object-oriented paradigm and its elements, discussing the advantages and disadvantages of the OO methodology. Students will delve into C++ fundamentals, including data types, operators, expressions, and control flow. Essential topics covered include arrays, strings, pointers, and functions, along with the creation and management of classes and objects. The course also addresses constructors and destructors, operator overloading, inheritance, virtual functions, and polymorphism to provide a comprehensive understanding of OOP principles in C++. File handling in C++ is another significant aspect of this course. Students will learn about console streams and console stream classes, including formatted and unformatted console I/O operations and manipulators. The course covers file streams, classes, file modes, file pointers, and file manipulations, as well as file input and output operations. Additionally, students will learn about exception handling in C++, enabling them to write robust and error-resistant programs.

The course also offers an introduction to Java, a versatile and widely-used programming language. Students will learn about Java's data types, variables, and arrays, as well as operators and control statements. The course covers the creation and management of classes, objects, and methods, along with key OOP concepts such as inheritance, packages, and interfaces. Exception handling, multithreaded programming, strings, and input/output operations are also discussed, providing students with the skills needed to develop efficient Java applications.

Course Outcomes:

At the completion of the course, a student will be able to:

1. Gain the basic knowledge on Object Oriented concepts and describe the differences between traditional imperative design and Object-oriented design.
2. Create & design applications using Object Oriented Programming Concepts
3. Explain class structures as fundamental, modular building blocks and explain the role of inheritance, polymorphism, dynamic binding and generic structures in building reusable code.
4. Write small/medium scale C++ / java programs with simple graphical user interface
5. Describe the file handling and error handling mechanisms in C++ and Create simple data structures like arrays in a Java program.
6. Describe to access databases through Java programs, using Java Database Connectivity (JDBC).

Acknowledgements:

The content we have utilized is solely educational in nature. The copyright proprietors of the materials reproduced in this book have been tracked down as much as possible. The editors apologize for any violation that may have happened, and they will be happy to rectify any such material in later versions of this book.

Unit : 1

Principles of OOP

Objective:

- Introduction of Principle of OOP's concept
- Understand Object Oriented Methodology
- Overview of procedure Oriented programming
- Definition of Object Oriented Programming.
- Object Oriented Languages”

Structure:

1.1 Software crisis
1.2 Software Evaluation
1.3 “POP” (Procedure Oriented Programming)
1.4 “OOP” (Object Oriented Programming)
1.5 Basic concepts of “OOP”
1.6 Benefits of “OOP”
1.7 Object Oriented Language
1.8 Application of “OOP”

Software Crisis

Developments in software technology are constantly evolving, with new tools and techniques being introduced at a rapid pace. This dynamic environment compels software engineers and the industry to continually seek innovative approaches to software design and development. The increasing complexity of software systems, coupled with the highly competitive nature of the industry, has made these new approaches even more critical. These rapid advancements have led to a perceived crisis within the industry. Several key issues need to be addressed to navigate this crisis effectively:

- How to accurately represent real-life entities of problems in system design?
- How to design systems with open interfaces?
- How to ensure the reusability and extensibility of modules?
- How to develop modules that can adapt to future changes?
- How to enhance software productivity and reduce costs?
- How to improve the quality of software?

  How to effectively manage time schedules?

Software Evaluation

Renowned artificial intelligence author Ernest Tello likened the development of software technology to the growing of a tree. Software has grown through multiple stages, or "layers," of development, much like a tree.

These layers have been built over the past 50 years, one layer on top of the other, each denoting a development over the one before it, as fig. 1.1 shows. However, the analogy breaks down when we consider how long these layers last. In a tree, only the top layer is active, but in a software system, all layers are active.

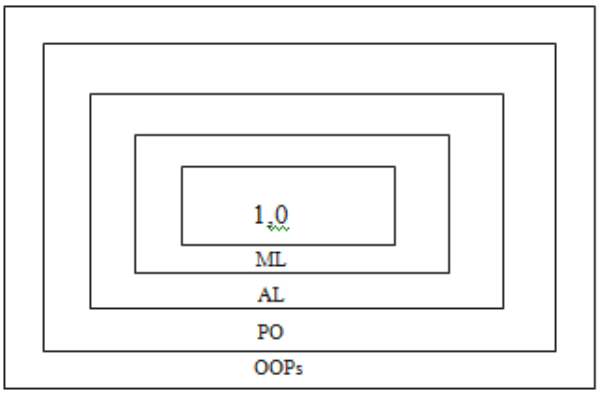


Fig.1.1 Software Layers of Growth

Architecture takes precedence over basic materials as complexity rises, according to Alan Kay, a proponent of the object-oriented paradigm and the main creator of Smalltalk. Building complicated software nowadays requires more than just stringing together sets of instructions, modules, and programming statements—we also need to use sound development practices and easily understood, implementable, and modifiable program structures.

The introduction of programming languages like C led to the rise in popularity of structured programming, which dominated the field in the 1980s. Programmers could develop somewhat complex programs with relative ease thanks to structured programming, a valuable technique. But as the programs got bigger, even the structured technique was unable to provide the requisite number of programs that were free of bugs.

Object Oriented Programming (OOP) is an approach to designing and developing programs that combines the best elements of structured programming with a range of innovative and potent concepts in an attempt to address some of the drawbacks of traditional programming techniques.

It is a fresh method of organizing and developing applications, not tied to any particular language. However, implementing OOP concepts just isn't possible with every language.

The term "object-oriented programming" refers to a programming approach that revolves around the ideas of classes, objects, inheritance, polymorphism, abstraction, encapsulation, and other related ideas.

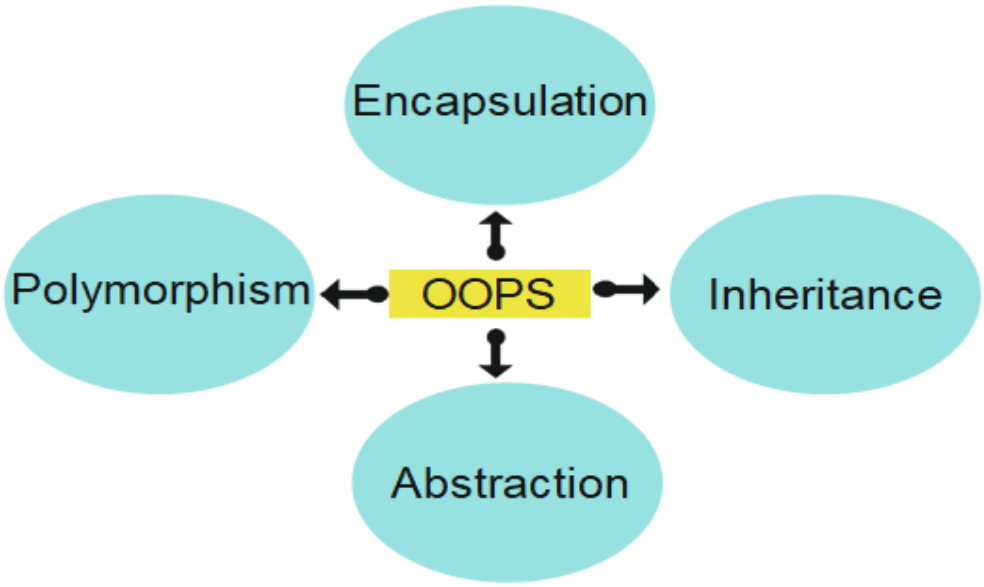


Fig.1.2: OOP's Module

1.1. Procedure-Oriented Programming

The tasks that need to be performed, such reading, calculating, and printing in languages like C, FORTRAN, and Bolt are viewed as the problem. The primary area of concern is functions. Figure 1.3 depicts a common procedural programming structure. The tasks required to solve a problem have been specified using the hierarchical decomposition technique.

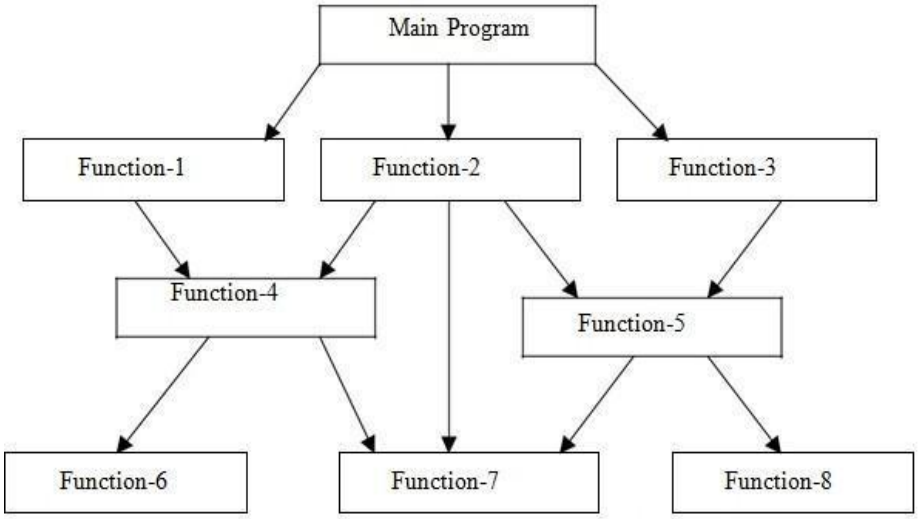


Fig.1.3 Structure of "Procedural Oriented Programs"

"Procedure-oriented programming" involves creating a sequence of instructions for the computer to follow and organizing these instructions into groups called functions.

Typically, flowcharts are used to arrange these actions and show how control moves from one to the next. In a multi-function software, a lot of important data items are set as global, meaning that every function can access them. Each function might include distinct local data. Global data are more prone to inadvertent changes by a function. Determining which function utilizes which data in a huge software is a very challenging task. Any functions that access the data must also be altered if an external data structure needs to be changed. This provides an opening for pests to enter.

We do not do a very good job of modeling real-world problems, which is another significant disadvantage of the procedural method. This is due to the fact that functions are action-oriented and do not truly correlate to the problem's element.

Procedure-oriented programming exhibits several key characteristics:

- The primary focus is on performing tasks (algorithms).
- Large programs are broken down into smaller, manageable units called functions.
- Many functions operate on shared global data.

- Data is passed openly from one function to another throughout the system.
- Functions are designed to transform data from one form to another.
- A “top-down approach” is used in the design of programs.

1.2. Object Oriented Paradigm “(OOP)”

Eliminating some of the procedural approach's shortcomings served as the primary driving force for the development of the object-oriented approach. OOP restricts the free flow of data across the system and views it as a vital component of program development. It safeguards data from inadvertent alteration by external functions and strengthens the link between the data and the functions that use it. With OOP, an issue can be divided into several things known as objects, and data and functions can then be built around these objects. Figure 1.3 illustrates how data and functions are organized in object-oriented applications. The function that is linked to an object is the only way to access its data.

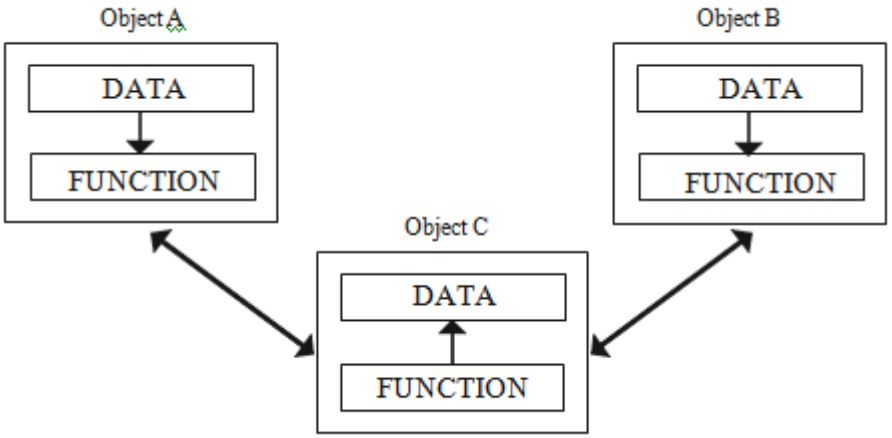


Fig 1.4: Organization of data and function in OOP

Object-oriented programming has several distinct features:

- “Emphasis is placed on data rather than procedures”.
- “Programs are organized into units called objects”.
- “Data structures are designed to represent objects”.
- “Functions that operate on an object's data are encapsulated within the object's data structure”.
- “Data is hidden and cannot be accessed by external functions”.
- “Objects can communicate with each other through functions”.
- “It is easy to add new data and functions as needed”.

- “The design process follows a bottom-up approach”.

The newest paradigm in programming, object-oriented programming, still has varied connotations for different individuals.

1.5 Basic Concepts of Object Oriented Programming

Some of the widely used principles in object-oriented programming must be understood. Among them are:

- “Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing”

1.5.1 Objects

In an object-oriented system, objects are the fundamental runtime elementsThey could represent any type of object that the application needs to work with, such as a person, place, bank account, or set of data.

Additionally, user-defined data like lists, vectors, and time may be represented by them. Analysis of programming problems is done in terms of objects and how they communicate with one another. It is important to select program objects that closely resemble real-world items. Like a record in Pascal or a structure in C, objects have an address and occupy space in the memory.

During program execution, messages are exchanged between the objects to facilitate interaction. For instance, in a software, if the objects "customer" and "account" are defined, the customer object might communicate with the count object to obtain the bank balance. Every object has code to alter data as well as data. Interactions between items do not require knowledge of each other's internal data or programming details. It is sufficient to know the types of messages that can be sent and the responses that objects return. Figure 1.5 illustrates two notations commonly used in object-oriented design and analysis, as depicted by various authors.

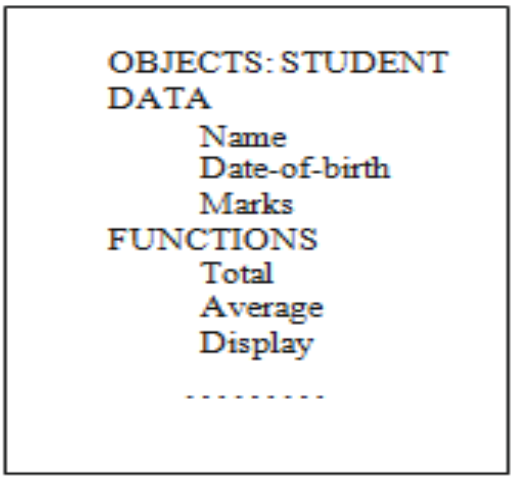


Fig. 1.5 An object representation

1.5.2 Classes

As we just discussed, objects are made up of code that manipulates data. With the use of classes, "an object's whole set of data and code can be converted into a user-defined data type". Objects are actually type class variables. We are able to generate an infinite number of objects that belong to a class after it has been defined. The data of the type class that each object is generated with is linked to it.

There is hence a variety of linked categories of goods. Members of the orange, mango, and apple fruit families are among them. Classes are user-defined types, much like a computer language's built-in sorts. The syntax for creating an integer object in C and the "Fruit Mango;" will produce a mango object that is a member of the fruit class.

1.5.3 Data Abstraction and Encapsulation

Encapsulation refers to "the bundling of data and functions into a single unit, known as a class". This is a prominent feature of a class, where the data is inaccessible to the outside world and can only be accessed through the functions contained within the class. These functions act as the interface between the object's data and the program. This insulation of data from direct access by the program is known as data hiding or information hiding .

"Abstraction involves representing essential features without including background details or explanations. Classes employ the concept of abstraction and are defined by a list of abstract attributes, such as size, weight, and cost, along with functions that operate on these attributes. They encapsulate all the essential properties of the object to be created.

Attributes are sometimes referred to as data members because they store information. The functions that operate on these data members are often called methods or member functions".

1.5.4 Inheritance

Inheritance is the "process by which objects of one class acquired the properties of objects of another classes". It supports the "concept of hierarchical classification". For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'. The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in fig 1.6.

In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes .

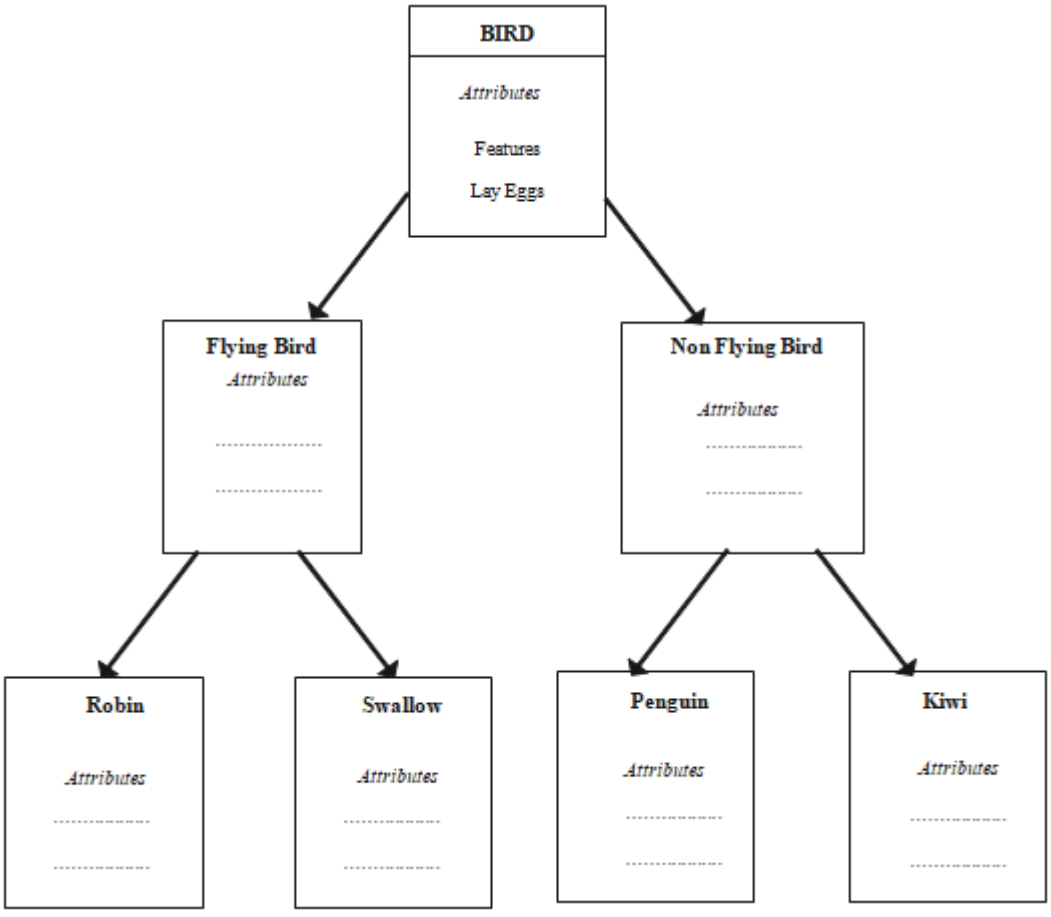


Fig. 1.6 "Property Inheritances"

1.5.5 Polymorphism

Another crucial idea in OOP is polymorphism. The Greek word polymorphism refers to the capacity to assume more than one form. Depending on the circumstances, an operation may behave differently. The kinds of data utilized in the process determine the behavior. For instance, think about how addition works. Two numbers will have their sum produced via the technique. If the operands were strings, concatenation would produce a third string. Operator overloading is "the process of forcing an operator to behave differently depending on the situation".

Figure 1.7 illustrates how different function names can handle different kinds and quantities of parameters. This is similar to how a single word may mean many things depending on the context. The practice of doing various tasks using a single function name is known as function overloading.

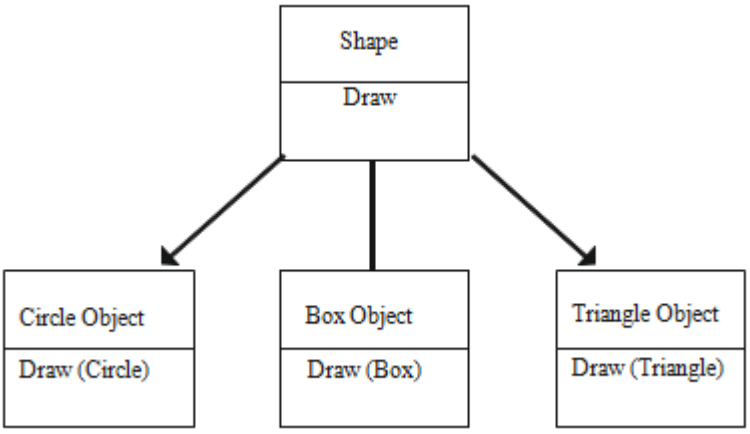


Fig. 1.7 Polymorphism

The ability for objects with distinct internal structures to share an external interface is largely due to polymorphism. This implies that although the particular action 8 linked to each operation may vary, a general class of operations may be obtained in the same way.

1.5.6 Dynamic Binding

Binding is "the process of linking a procedure call to the code that will be executed in response to the call. When using dynamic binding, a procedure call's matching code is not known until the call is made during runtime". It has to do with polymorphism and

inheritance. The dynamic type of a polymorphic reference determines the function call associated with it.

Take a look at the "draw" process in fig. 1.7. Every object will have this process by inheritance. The draw procedure will be rewritten in each class that defines the object because its algorithm is specific to each object. The code corresponding to the object now referenced will be called at runtime.

1.5.7 Message Passing

An object-oriented program comprises "a collection of objects that interact with each other". Programming in an object-oriented language involves the following fundamental steps:

1. "Creating classes that define objects and their behaviors".
2. "Instantiating objects from these class definitions".
3. "Establishing communication among objects".

Objects communicate by sending and receiving information, much like people exchanging messages. This concept of message passing simplifies the construction of systems that directly model or simulate real-world entities.

A message sent to an object is essentially a request for the execution of a procedure, invoking a function (procedure) within the receiving object to produce the desired results. Message passing involves specifying the name of the object, the name of the function (message), and the information to be sent. Example:

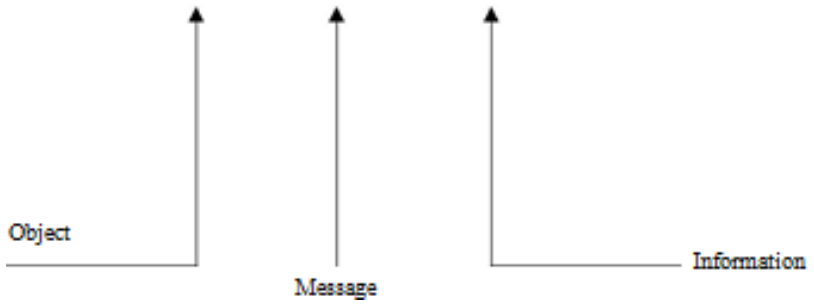


Fig1.8 : Employee Salary

An object's life cycle exists. It is possible to construct and destroy them. As long as an object is alive, communication with it is possible.

1.6 Benefits of OOP

OOP's provides the user and program designer with a number of advantages. Numerous issues pertaining to the creation and caliber of software products are resolved in part by object-orientationThe new technology promises reduced maintenance costs, more programmer efficiency, and better software quality.

The main benefits are

- By using inheritance, we can extend the use of pre-existing classes and remove unnecessary code.Instead of starting from scratch while developing the code, we can create programs that interact with one another using the standard working modules. Higher productivity and the saving of development time result from this.
- By applying the data hiding concept, programmers can design safe programs that are impenetrable to code from other areas of the program.
- Objects in the issue domain can be mapped to object in the program, and multiple
- instances of an object can exist side by side without causing conflicts
- .It is easy to divide up a project's work by objects.
- We are able to capture more information about a model's implemental form thanks to the data-centered design method.
- Upgrading object-oriented systems from small to large systems is simple.
- Interface specifications with external systems become easier to interface with thanks to message passing techniques for communication between objects.
- It is simple to control software complexity.

"All of these capabilities can be included in an object-oriented system, but how important they are will depend on the project's nature and the programmer's preferences. To enjoy some of the aforementioned advantages, a number of problems must be resolved. Object libraries, for example, have to be reusable. Since technology is always evolving, the existing product could soon become outdated. If it is not to be compromised, certain protocols and controls must be created".

1.7 Object Oriented Language

No specific language has the right to object-oriented programming. Similar to structured programming, OOP ideas can be implemented in Pascal and C. However, when programs get bigger, they become awkward and might cause confusion. It is simpler to apply OOP concepts in a language that has been specifically created to accommodate them.

For a language to be considered object-oriented, it must support many OOP principles. They can be divided into the following two groups based on the characteristics they support:

- "Object-based programming languages", and
- "Object-oriented programming languages".

The primary programming paradigm that facilitates encapsulation and object identification is called object-based programming. Key components needed for object-based programming include:

- "Data encapsulation"
- "Data hiding and access mechanisms"
- " Automatic initialization and clear-up of objects"
- "Operator overloading"

Programming languages that facilitate object-based programming are referred to as objects-based programming languages. Both dynamic binding and inheritance are not supported by them. One common object-based programming language is Ada.

All of the elements of object-based programming are included in object-oriented programming languages, coupled with two more aspects: inheritance and dynamic binding. Consequently, the following claims describe object-oriented programming:

Object-based features + inheritance + dynamic binding

1.8 Application of OOP

The newest in programming is OOP. OOP seems to be generating a lot of excitement and interest among software engineers. OOP applications are starting to become more significant in a variety of fields. Up until now, the most widely used application of object-oriented programming has been in the design of user interfaces, like windows. Using OOP approaches, hundreds of windowing systems have been created.

These days, OOP is a common programming term. OOP looks to be a topic that software engineers are really excited and interested in using. Using OOP is starting to become more significant in a lot of fields. Up until now, object-oriented programming has been most commonly used in the design of user interfaces, such as windows. OOP approaches have been used to construct hundreds of windowing systems.

- Real-time system

- Simulation and modeling
- Object-oriented data bases
- Hypertext, Hypermedia, and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

The language gave rise to the object-oriented paradigm, which developed into design and, more recently, analysis. It is thought that the OOP environment's richness will help the software sector increase software system productivity as well as quality.

Summary

Object-oriented programming (OOP) offers numerous advantages to both program designers and users. It addresses many challenges associated with software development and enhances the quality of software products. This modern approach promises increased programmer productivity, improved software quality, and reduced maintenance costs. Consequently, traditional programming languages need to evolve to incorporate object-oriented concepts, as they are more intuitive for humans to understand. These benefits are particularly evident when learning languages like C++.

Software technology has undergone significant evolution over the past five decades. Procedural Oriented Programming (POP) adopts a top-down approach, treating the problem as a sequence of tasks to be executed with functions written to perform these tasks. However, "POP" has two primary drawbacks: "data can move freely within the program, and it does not model real-world problems effectively".

Object-Oriented Programming (OOP) was developed to address these drawbacks. OOP follows a bottom-up approach, considering the problem as a collection of objects where objects are instances of classes. Data abstraction is a key concept in OOP, focusing on essential features without including unnecessary details. Inheritance allows objects of one class to acquire properties of objects from another class. Polymorphism enables the use of the same name for multiple functions within a program. Dynamic binding ensures that the code associated with a procedure is determined at runtime.

Self Assessment Questions

Object-Oriented Programming (OOP)

1. What is encapsulation, and why is it important in OOP?
2. Explain the concept of inheritance and its advantages.
3. What is polymorphism in OOP? Provide an example.
4. Describe the difference between a class and an object.
5. What is the purpose of an interface in OOP? How does it differ from a class?
6. How does message passing work in OOP?
7. What is abstraction, and how does it differ from encapsulation?
8. Explain the concept of data hiding. Why is it used in OOP?
9. What are constructors and destructors? How are they used in OOP?
10 What is method overloading and method overriding? Provide examples for each.**

Procedure-Oriented Programming (POP)

1. What is the primary focus of Procedure-Oriented Programming?
2. How are large programs managed in POP?
3. Explain the concept of global data in POP and its implications.
4. Describe the top-down approach used in POP.
5. What are the main disadvantages of using POP compared to OOP?
6. How is data shared between functions in POP?
7. What is a procedure or function in POP, and how is it used?
8. Compare and contrast the error handling mechanisms in POP and OOP.
9. How does the reusability of code in POP differ from that in OOP?
10 Explain how program design differs between POP and OOP.

Unit - 2

Basics of C++

Objective:

- A Brief History of C and C++
- Difference between C and C++
- Features of C++
- Advantages and Disadvantages of C++
- Applications of C++
- Writing and Executing a C++ Program
- Program Structure and Rules
- Sample C++ Program
- Comments
- Return Type of MAIN()
- Namespace std
- Header File
- Output Statement (COUT)
- Input Statement (CIN)

Structure:

2.1. A Brief History of C and C++

C is a versatile, procedural, and imperative programming language created in 1972 by Dennis M. Ritchie at Bell Telephone Laboratories for the development of the UNIX operating system. Over the years, C has become one of the most widely adopted programming languages, frequently competing with Java for the top spot in terms of popularity among software developers.

C++ is an object-oriented programming language that was designed by Bjarne Stroustrup in the early 1980s at AT&T Bell Laboratories in Murray Hill, New Jersey, USA. Stroustrup, who was inspired by the language Simula67 and a strong advocate for C, aimed to merge the best aspects of both languages to create a more powerful tool that supported object-oriented programming while maintaining the strengths of C. This led to the creation of C++, an extension of C that incorporates the class construct from Simula67. Initially referred to as "C with classes," the language was renamed C++ in 1983, drawing from the C increment operator "++" to signify that C++ is an enhanced version of C.

C++ serves as a superset of C, meaning that almost all programs written in C can also be compiled in C++. However, there are some minor differences that might prevent certain C programs from running seamlessly under a C++ compiler.

The most significant additions that C++ brings to C include classes, inheritance, function overloading, and operator overloading. These features allow for the creation of abstract data types, the ability to inherit properties from existing data types, and support for polymorphism, solidifying C++ as a robust object-oriented programming language.

2.2 Difference between C and C++

C and C++ are two distinct programming languages. C is known for its procedural programming approach, whereas C++ incorporates object-oriented programming principles. C++ was developed to address some of the limitations found in C.

The primary distinction between C and C++ lies in their programming paradigms. C is purely procedural and does not support classes or objects. On the other hand, C++ blends both procedural and object-oriented programming features, making it a hybrid language.

The detailed differences between C and C++ are shown in the following table.

Aspect	C	C++
Language Paradigm	Procedural	Multi-paradigm (procedural, object-oriented, generic, functional)
Origin	Developed in 1972 by Dennis Ritchie	Developed in 1983 by Bjarne Stroustrup as an extension of C
Standard Libraries	Standard Library provides basic functions	Standard Library includes the Standard Template Library (STL) with containers, algorithms, and iterators
Syntax and Keywords	Basic set of keywords	Extended set of keywords (e.g., class, namespace, template)
Data Encapsulation	Absence of integrated encapsulation support	allows for encapsulation using classes and access specifiers (public, protected, and private).
Inheritance	opposes inheritance	allows for inheritance (hybrid, multiple, and single).
Polymorphism	No built-in support for polymorphism	Supports polymorphism (function overloading, operator overloading, virtual functions)
Memory Management	Manual memory management using malloc and free	Manual memory management using new and delete, as well as smart pointers for automatic management
Exception Handling	Error handling through return codes and errno	Built-in exception handling using try, catch, and throw
Namespaces	No support for namespaces	Supports namespaces to avoid name conflicts
Function Overloading	Does not support function overloading	Supports function overloading
Operator Overloading	Does not support operator overloading	Supports operator overloading
Inline Functions	Supports inline functions with static inline	Supports inline functions using the inline keyword
Templates	Does not support templates	Supports templates for generic programming
Standard Input/Output	Uses printf, scanf, etc. for input/output	Uses iostream library with cout, cin, etc.
File Extensions	Typically uses .c file extension	Typically uses .cpp or .cc file extension
Complex Data Types	Uses structures (struct) for complex data types	Uses classes (class) and structures (struct), with classes supporting more features like inheritance and access control
Object-Oriented Features	No support for object-oriented features	Supports object-oriented features (encapsulation, inheritance, polymorphism)

Aspect	C	C++
Default Arguments	Does not support default arguments in functions	Supports default arguments in functions
STL (Standard Template Library)	No STL	Includes STL for data structures and algorithms
Type Safety	Less type-safe, more prone to errors	More type-safe due to stricter type checking and features like references
Virtual Functions	Does not support virtual functions	Supports virtual functions for dynamic polymorphism
Constructors/Destructors	No constructors or destructors	Supports constructors and destructors for object lifecycle management
Friend Functions	Does not support friend functions	Supports friend functions for class access
Const Keyword	const keyword for constant variables	Enhanced use of const for variables, pointers, and member functions
Preprocessor Directives	Extensively uses preprocessor directives (#define, #include)	Uses preprocessor directives but provides better alternatives (e.g., templates)
Compilation	Generally faster due to simpler syntax and fewer features	Typically slower due to complex features and additional checks
Portability	Highly portable across different systems	Portable but may require dealing with more complex compiler dependencies

2.3 Features of C++

C++ is a highly adaptable language designed to manage very large-scale programs and is suitable for a wide range of programming tasks. These include developing editors, compilers, databases, communication systems, and other complex real-life application systems.

- C++ allows the creation of hierarchically related objects, enabling the development of specialized object-oriented libraries that can be reused by multiple programmers.
- The combination of object-oriented and procedural features in C++ allows it to effectively model real-world problems while also providing the low-level programming capabilities of C.
- C++ programs are known for their maintainability and scalability. New features can be easily integrated into the existing structure of an object.
- It is anticipated that C++ will eventually become the preferred general-purpose programming language, surpassing C.

2.4 Advantages and Disadvantages of C++

Advantages of C++:

1. **Object-Oriented Programming:** C++ supports object-oriented principles like inheritance, encapsulation, and polymorphism, which promote code reusability and modularity.
2. **Efficiency and Performance:** C++ is a compiled language that provides high performance and efficient memory management, making it ideal for applications where speed and resource optimization are crucial.
3. **Rich Library Support:** The Standard Template Library (STL) in C++ offers a variety of pre-built classes and functions for data structures and algorithms, which simplifies the development process.
4. **Portability:** C++ programs can be run on various platforms with little to no modification, making it a portable language.
5. **Flexibility:** C++ allows both low-level and high-level programming. This flexibility makes it suitable for system-level programming as well as application development.
6. **Scalability:** C++ is well-suited for large-scale software development projects, providing tools to manage and organize large codebases effectively.

Disadvantages of C++:

1. **Complexity:** The complexity of C++ syntax and its numerous features can make the language difficult to learn and master, especially for beginners.
2. **Manual Memory Management:** While providing control over memory allocation and deallocation, manual memory management in C++ can lead to issues like memory leaks and buffer overflows if not handled correctly.
3. **Lack of Garbage Collection:** Unlike some modern programming languages, C++ does not have built-in garbage collection, requiring developers to manage memory manually, which can increase the risk of memory-related errors.
4. **Long Compilation Times:** The compilation process in C++ can be time-consuming, especially for large projects, which can slow down the development cycle.
5. **Security Issues:** The powerful features of C++, such as pointer arithmetic and direct memory access, can lead to security vulnerabilities if not used carefully.

2.5 Applications of C++

The C++ language is mostly used to develop system software and desktop applications. Below are a few examples of C++ language applications.

To develop applications connected to graphics, such as computer and mobile gaming. ·Use the C++ language to evaluate any type of mathematical equation. Similar to Windows XP, OS design also uses the C++ language. C++ is the programming language used to write X. Internet browser Firefox is written in C++ programming language. The C++ programming language is used to develop all of Adobe Systems' main apps, such as Adobe Premier, Illustrator, Photoshop, and ImageReady. A few Google apps, such as Google Chromium and the Google File System, are also written in C++. MySQL-like database design is done with C++.

2.6 Writing and Executing a C++ Program

TC installation is quite easy. Simply download and run.exe files to use Turbo C or C++. A TC directory and other subdirectories, like INCLUDE and LIB, are generated on the hard drive when you install the Turbo C or C++ compiler on your computer. (Figure 2.1)

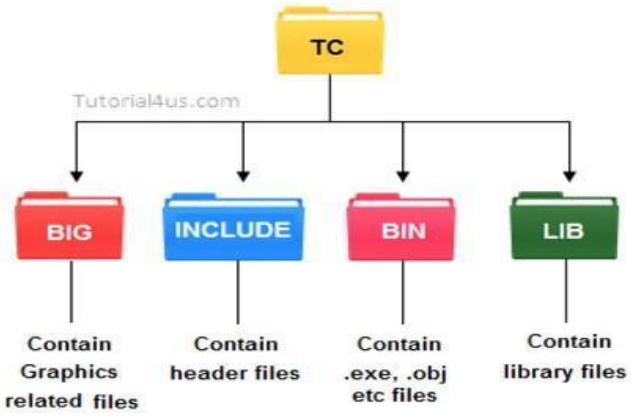


Figure 2.1 : Turbo C

- **INCLUDE**: Houses the header files for both C and C++.
- **LIB**: Stores the library files used by C and C++.
- **BGI**: Contains files related to graphics.
- **BIN**: Holds executable files (.exe), object files (.obj), and others.

TC Editor: TC Editor is incredibly straightforward and user-friendly; in this post, I'll provide you with all of my TC Editor tips as well as some shortcut keys that come in handy when coding. The most popular C language compiler is called Turbo C. I'll go over every aspect of

its interface below. Turbo C has a fairly straightforward user interface. The navigation bar is active when the IDE screen displays. It has a number of menus, including:

File: A collection of commands for saving, editing, printing programs, and exiting the Turbo C editor are contained in this menu.

Edit: A collection of commands for altering the source code of C programs are available on this menu. For instance, cut, paste, undo, and copy.

Search: This menu includes a set of commands that can be used to look up a certain word and swap it out with another.

Run: A collection of commands for executing C programs can be found in this menu.

Compile: A collection of commands used to compile C programs are contained in this menu.

Debug: A collection of commands for troubleshooting C programs are located in this menu.

Project: A collection of commands for starting, stopping, and creating projects are located on this menu.

Options: This menu includes a collection of commands for configuring the Turbo C IDE, creating directories, and other things.

Windows: A collection of commands for opening and shutting different IDE windows are located in this menu.

Help: Use this option to obtain assistance with a particular C language topic. Likewise, to obtain assistance with a certain keyword or C identifier.

Shortcut Keys for TC Editor:

- **Alt + X**: Exit the TC Editor.
- **Ctrl + F9**: Execute the C program.
- **Alt + F9**: Compile the C code.
- **Alt + Enter**: Toggle between full screen and windowed mode.
- **Ctrl + Y**: Delete the entire line above the cursor.
- **Shift + Right Arrow**: Select a line of code.
- **Ctrl + Insert**: Copy selected text.
- **Shift + Insert**: Paste copied text.
- **Shift + Delete**: Delete selected text.

Figure 2.2 TC Editor

2.7 Program Structure and Rules

Similar to C, a C++ program consists of a collection of functions. The example above includes only one function, main(). Execution starts at the main() function, which is a requirement for every C++ program. With a few notable exceptions, the compiler disregards line breaks and whitespace in C++ as it is a free-form language. Statements in C++ are ended with semicolons, much like in C.

2.8 A sample C++ Program

In this guide, I will present C++ basic programs in a straightforward and easy-to-understand manner. The coding examples will be done in C++ and will be accompanied by pictures and relatable real-life scenarios. Our goal is to provide the best and simplest tips, tricks, and techniques for programming. We hope this guide is beneficial for all our visitors in learning C++.

Being a better skilled programmer is the goal of learning a programming language. This entails developing more proficiency in both creating and executing new systems as well as maintaining current ones.

Writing in a manner akin to Fortran, C, Smalltalk, etc. is possible with C++'s support for several programming paradigms, all while preserving runtime and space efficiency. Hundreds of thousands of programmers in almost every application domain use C++. Writing device drivers and other software that has to directly manipulate hardware while operating under time limits is one area where it is very common. Additionally, because of its clarity, which

makes it useful for teaching foundational topics, C++ is frequently used in research and education. Because C++ is used to write these systems' main user interfaces, whether you've used an Apple Macintosh or a PC running Windows, you've indirectly dealt with the language.

Let's start with a simple C++ program example that prints a string to the screen:

```
“#include<iostream>
Using  namespace std;
int main()
        {
        cout<<”c++ is better than c \n”;
        return 0;
        }
```

This simple program demonstrates several C++ features.

2.9 Comments:

A new comment sign, // (double slash), is introduced in C++. Double slashes at the beginning of comments carry through to the end of the line. They can begin anywhere in the line, and anything following the double slash is ignored by the compiler. Note that there is no closing symbol for this type of comment.

The double slash is used for single-line comments. For example:

```
// This is an example of
// a C++ program to illustrate
// some of its features
```

For multiline comments, the traditional C comment symbols /* and */ are still valid and often more suitable. For instance:

```
/* This is an example of a C++ program
   to illustrate some of its features */
```

The only statement in the sample program 1.10.1 is an output statement:

```
cout << "C++ is better than C.";
```

This statement causes the string within the quotation marks to be displayed on the screen. It introduces two new C++ features: **cout** and **<<.**

The identifier **cout** (pronounced "C out") is a predefined object that represents the standard output stream in C++. Typically, this standard output stream is the screen, but it can be

redirected to other output devices. The << operator is known as the insertion operator or the "put to" operator.

2.10 Return Type of MAIN( ):

The operating system receives an integer value from C++'s main() function. As a result, in C++, each main() function should terminate with a return (0) statement; otherwise, an error warning can appear. The type of main() is expressly defined as int as the function returns an integer. Remember that all C++ functions have int as their default return type. If type and return are not provided, the following main will run with an error message:

```
main ()
{
    ……
    - - - - - - - - -
}
```

2.11. Namespace std:

Namespaces are a notion that the ANSI C++ standards committee established to provide a program's identifiers a scope. The using directive is used, such as this, to make use of the identifiers declared within a namespace:

using namespace std;

The namespace where the ANSI C++ standard class libraries are declared is represented by the symbol std in this line. This directive must be incorporated into all ANSI C++ applications. With the help of this directive, the current global scope now includes all of the identifiers declared in std. The C++ language has included two new keywords: using and namespace.

2.12. Header File:

Header files are included at the beginning of a C++ program, similar to C programs. For instance, **iostream** is a header file that provides input and output streams. These header files contain pre-declared function libraries, which users can utilize for their convenience. To include the **iostream** header file in the program, we use the following **#include** directive:

#include <iostream>

The #include directive tells the compiler to add the contents—separated by angle brackets—of the given file to the source file. All applications that include input/output operations should start with the iostream header file.

An input statement that tells the software to wait for the user to enter a number is cin >> number1;. The number that was entered is then kept in the variable number 1. The standard input stream, which is normally represented by the predefined object cin (pronounced "C in") in C++, is where input is normally entered via the keyboard.

The >> operator is referred to as the extraction or "get from" operator. It extracts (or retrieves) the value from the keyboard and assigns it to the variable on its right-hand side. This operation is akin to the familiar **scanf()** function in C. Similar to **<<,** the >> operator can also be overloaded.

2.13. Output Statement (COUT)

The insertion operator \\ has been used to print results many times in the last two sentences.

The assertion

cout << "Sum = " << sum << "\n";

Sends the value of sum to cout after sending the string "Sum = " first. In order to guarantee that the next output starts on a new line, it appends the newline character. Cascade refers to the numerous usage of \\ in a single phrase. It is imperative to incorporate the appropriate blank spaces in between items when cascading output operators.

Combining the final two sentences using the cascading approach yields the following result:

```
cout << "Sum = " << sum << "\n"
    << "Average = " << average << "\n";
```

This single statement provides two lines of output. If you prefer both outputs on one line, you can use:

```
cout << "Sum = " << sum << ", "
    << "Average = " << average << "\n";
```

This will produce output like:

Sum = 14, Average = 7

cout << is used to display content on the screen, similar to **printf** in C++. **cin** and **cout** are akin to **scanf** and **printf**, but without the need for format specifiers like **%d** for integers; in **cout** and **cin**, the appropriate data types are inferred automatically.

2.14. Input Statement (CIN)

As seen below, we may also cascade input operator >>

From left to right, the values are allocated as Cin >> number1 >> number2. In other words, if we enter two values, let's say 10 and 20, number 1 will receive 10 and number 2 will receive 20.

2.15. Text Editor

A variety of text editors are available for typing your program. Examples include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

The specific name and version of the text editor may vary depending on the operating system. For instance, Notepad is commonly used on Windows, while vim or vi can be utilized on both Windows and Unix-based systems like Linux.

The files created using your chosen editor are referred to as source files, and for C++, they typically have extensions such as .cpp, .cp, or .c.

Having a text editor in place is essential to commence your C++ programming journey.

2.16. C++ Compiler:

Your source code will be compiled into an executable application using this real C++ compiler. The extension you give your source code doesn't matter to most C++ compilers, but many will use.cpp by default if you don't indicate otherwise.

The most popular and freely accessible compiler is the GNU C/C++ compiler; if you have the corresponding operating system, you may also use compilers from HP or Solaris.

2.17. Summary:

This chapter provides you a glimpse of C++. So far we have learned the concept of Object Oriented Programming paradigm and in this chapter we have the concepts of C++ and the advantages and disadvantages of programming in C++. Eventually, we have learned the basics of writing programs using C++ and introduction of an editor and its methods. We have learned the basics of installation of C++ editor and writing the program and the requirements of programming in C++. Ordinarily, so far we have not yet started writing programs using OOPs programming and understanding of writing programs with Object Oriented Programming with C++.

The idea of functions in C++ has been discussed, including both declaration and definition. We have also explored the notion of classes, including their definition and declaration. In

addition, we have explained how to create objects and get the data members of a class. Furthermore, we have shown how to use call by reference and call by value approaches to send objects as parameters to functions.

2.18. Self-Assessment Questions:

1. What are the main features of the C++ programming language?
2. Explain the basic structure of a C++ program.
3. How do you define a variable in C++? Provide examples.
4. What are the different data types available in C++?
5. Describe the concept of pointers in C++. How are they used?
6. How does C++ handle input and output operations?
7. What is the purpose of including the iostream header in a C++ program?
8. What are the different operators available in C++ and how are they used?
9. Explain the concept of functions in C++. How do you declare and define a function?
10. What is the difference between pass-by-value and pass-by-reference in C++?
11. Describe the concept of classes and objects in C++. How are they declared and used?
12. What is the significance of constructors and destructors in C++? How are they defined and when are they called?

Unit - 3

Expression

Objective:

- Introduction C++ Tokens
- Understand different Data types in C++
- Understand Declaration of Variables
- How to Initialization of Variables in C++
- Understand Reference Variables
- Know the Operators and use of the Operators in C++
- Type Cast Operator in C++
- Understand Memory Management operators and use of the same
- Mentioning of Expression
- Understand Statement
- Understand Symbolic Constant
- Type Compatibility with C++

Structure:

3.1 Introduction to C++ Tokens

In the realm of programming languages, a C++ program can be described as an arrangement of entities that interact by invoking each other's functions. This chapter delves into the utilization of Object-Oriented Concepts in C++, specifically focusing on the concepts of classes, objects, methods, and instance variables.

- **Object**: Objects are entities that possess both state and behavior. For instance, a dog can be characterized by its states such as color, name, and breed.
- **Class**: A class serves as a blueprint
- **Methods**: Methods encapsulate behaviors within a class. A class can comprise multiple methods, where the implementation of functionalities, manipulation of data, and execution of actions are carried out.
- **Instance Variables**: Each object maintains a distinct set of instance variables, contributing to its unique state. These instance variables are instrumental in defining an object's characteristics and are assigned specific values to delineate its state.

By comprehending these fundamental concepts, one can construct and manipulate objects within a C++ program efficiently, facilitating the implementation of Object-Oriented Principles in software development.

C++ Program Structure

Below is a simple C++ code snippet that prints the words "Hello World":

```
#include <iostream>
using namespace std;
// main() is where program execution begins.
int main() {
cout << "Hello World"; //prints Hello World
return 0;
}
```

Fig 3.1 Hello Program

This program could be explained as: −

- The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header <iostream> is needed.

- The line using namespace std; tells the compiler to use the std namespace.

Namespaces are a relatively recent addition to C++.

- The next line '// main() is where program execution begins.' is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
- The line int main() is the main function where program execution begins.
- The next line cout << "Hello World"; causes the message "Hello World" to be displayed on the screen.
- The next line return 0; terminates the main( )function and causes it to return the value 0 to the calling process.

3.1.1 Compile and Execute C++ Program

To compile and run a C++ program, follow these steps:

1. Open a text editor and input the code as described in the previous section.
2. Save the file as: hello.cpp.
3. Open a command prompt and navigate to the directory where you saved the file.
4. Type the command program and press Enter to compile your code. If there are no errors, the command prompt will proceed to the next line and generate an executable file.
5. Now, execute the program by typing its name and running it.
6. You should see "Hello World" printed on the console window.

3.2 Understand Different Data Types in C++

In C++, data types are crucial for defining the type of variables and their contents. They dictate how storage is utilized in your programs and can be categorized into two types:

1. **Built-in Data Types**: These data types are predefined and integrated directly into the compiler. Examples include int, char, etc.

3.2.1 Built-in Data Types

Built-in data types are those that are predefined in the C++ language and are directly supported by the compiler. Examples of built-in data types include int, char, float, double, etc.

These data types provide the basic building blocks for defining variables and manipulating data within C++ programs. They are fundamental for performing arithmetic operations, storing characters, and managing memory efficiently.

Understanding built-in data types is essential for writing robust and efficient C++ code, as they form the foundation upon which more complex data structures and operations are built.

Basic Built in Data types in C++

char	for character storage (1 byte)
int	for integral number (2 byte)
float	single precision floating point (4 byte)
double	double precision floating point numbers (4 byte)

Fig 3.2 Data Types in C++

3.2.2. User defined

a. Abstract data types

Variables in C++ are utilized to store values that may change during program execution. They can be declared in various ways, each with distinct memory requirements and functionality. A variable represents a memory location allocated by the compiler based on its data type.

b. Enum as Data type in C++

Enumerated type declares a new type-name along with a sequence of values containing identifiers which has values starting from 0 and incrementing by 1 every time.

For Example: enum day(mon, tues, wed, thurs, fri) d;

Here an enumeration of days is defined which is represented by the variable d. mon will hold value 0, tue will have 1 and so on. We can also explicitly assign values, like, enum day(mon, tue=7, wed);. Here, mon will be 0, tue will be assigned 7, so wed will get value 8.

c. Modifiers in C++

In C++, special words called modifiers, can be used to modify the meaning of the predefined built-in data types and expand them to a much larger set. There are four datatype modifiers in C++, they are:

1. long

2. short
3. signed
4. unsigned

The above mentioned modifiers can be used along with built in data types to make them more precise and even expand their range.

Remember, below mentioned are some important points you must know about the modifiers:

- enum day(mon, tues, wed, thurs, fri) d;long and short modify the maximum and minimum values that a data type will hold.
- A plain int must have a minimum size of short.
- Size hierarchy : short int < int < long int
- Size hierarchy for floating point numbers is : float < double < long double
- long float is not a legal type and there are no short floating point numbers.
- Signed types includes both positive and negative numbers and is the default type.
- Unsigned, numbers are always without any sign, that is always positive.

3.3 What are Variables?

In C++, variables serve as storage containers for values that may change during program execution. They are declared in various ways, each with distinct memory requirements and functionality. A variable essentially represents a named memory location allocated by the compiler, contingent upon the data type of the variable.

Variables play a crucial role in programming as they enable the manipulation and storage of data, facilitating dynamic behavior within a program. By utilizing variables, programmers can store and retrieve values, perform calculations, and control the flow of program execution.

3.3.1 Basic Types of Variables

In C++, every variable declaration necessitates the specification of a data type, which determines the memory allocated to the variable. Here are the fundamental types of variables:

bool	for variable to store bloolean values (true or false)
char	for variables to store character types
int	for variables with integral values
float and double are also type for variables with large and floating point values	

Fig 3.3 Variables Types

3.3.2. "Declaration" and "Initialization"

Variables must be declared before use, and preferably at the beginning of the program. However, in C++, they can be declared anywhere before use. Initialization involves assigning a value to a declared variable.

```
int i;      //declared but not initialised
char c;
int i, j, k; // Multiple declaration
Initialization means assigning value to an already declared variable,
int i;  // declaration
i= 10; // initialization
Initialization and declaration can be done in one single step also,
int i=10;       //initialization and declaration in same step
int i=10, j=11;
```

If a variable is declared and not initialized by default it will hold a garbage value. Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

```
int i,j;
i=10;
```

For example:

```
int i;                          // declared but not
initialised char c;
int l, k;  // Multiple declaration
Initialization means assigning value to an already declared variable, int i;  //
declaration
1= 10;
Initialization and declaration can in one single step also,
int i=10;       //initialization and declaration in same step
int i=10, j=11;
```

If a variable is declared and not initialized by default it will hold a garbage value. Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

```
int i,j;
i=10;

j=10;
int j=i+j;  //compile time error, cannot redeclare a variable in same scope
```

3.3.3 Scope Of Variable

All variables in C++ have a scope, which defines the area in which they are functional. Variables exist within their scope and lose their value outside of it. This boundary is termed the "scope of the variable" and typically extends between curly braces {} where the variable is declared.

Variables in C++ can broadly be categorized into two main types: Global Variables and Local Variables.

Global Variables

Global variables are declared outside any function and can be accessed throughout the lifetime of the program by any function or class. They are not confined to a specific function's scope. Even if only declared without initialization, global variables can be assigned different values at different times during the program's execution. If initialized outside the main() function, they can be reassigned at any point in the program.

For example: Only declared, not initialized

```
include <iostream>
using namespace std;
 int x;              // Global variable declared
int main()
{
x=10;               // Initialized once
cout <<"first value of x = "<< x;
```

Local Variables

Local variables in C++ are restricted to the block in which they are declared, typically delimited by curly braces {}. Once outside of this block, they become inaccessible, leading to a compile-time error if attempted to access.

3.4 C++ Identifiers

In C++, an identifier serves as a name used to distinguish variables, functions, classes, modules, or any other user-defined entities. These identifiers follow certain rules:

- An identifier starts with a letter A to Z or a to z, or an underscore (_), followed by zero or more letters, underscores, and digits (0 to 9).
- C++ does not permit the use of punctuation characters such as @, $, and % within identifiers.
- C++ is a case-sensitive programming language, meaning that identifiers with different capitalization are treated as distinct entities. For example, Manpower and manpower are two separate identifiers in C++.

Here are some examples of acceptable identifiers:

- variable
- Function
- my_variable
- myFunction
- _myVariable
- MyClass
- MY_CONSTANT

- myVariable123

These identifiers adhere to the rules specified for naming conventions in C++, ensuring clarity and consistency in code readability and maintenance.

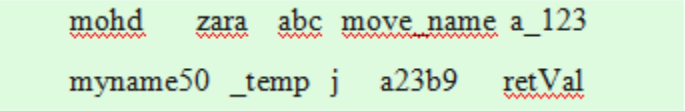


Fig 3.4 suitable Identifiers

3.5 C++ Keywords & Identifiers

C++ has a set of reserved keywords that are part of its syntax and cannot be used as identifiers (e.g., variable names, function names). These keywords have special meanings defined by the language. Below is a list of C++ keywords:

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

Fig 3.4 Reserved Keywords

3.6 Trigraphs

A trigraph sequence is an alternate form for a few characters. A trigraph is a sequence of three characters that always begins with two question marks and represents a single character. Trigraphs are expanded in comments, preprocessor directives, string literals, and character literals, among other places.

The most commonly used trigraph sequences are shown below –

Trigraph	Replacement
??=	#
??/	\
?? '	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

Fig 3.5 Trigraph

Trigraphs are not supported by any compiler and are not recommended for use due to their confusing nature

3.7 Whitespaces in C++

Whitespace in C++ refers to blanks, tabs, newline characters, and comments. These characters are used to separate parts of a statement and aid the compiler in identifying where one element ends and the next begins. A line consisting solely of whitespace, possibly with a comment, is termed a blank line, and the C++ compiler completely disregards it.

For example, in the statement:

```
int age;
```

There must be at least one whitespace character (usually a space) between int and age for the compiler to distinguish them.

In another example:

fruit = apples + oranges; // Get the total fruit

No whitespace characters are necessary between fruit and =, or between = and apples, although they can be included for readability.

3.8 Semicolons and Blocks in C++

In C++, the semicolon (;) is used as a statement terminator. Each individual statement must end with a semicolon, indicating the completion of one logical entity.

A block in C++ is a set of logically connected statements enclosed within opening and closing braces {}.

3.9 Summary

In this lesson, we covered the concept and types of constructors and destructors, along with operator overloading. Despite overloading operator functions, the precedence of operators remains unchanged. The ++ and -- operators can function as postfix or prefix operators, necessitating separate functions for overloading them for different applications. Additionally, private data of a class can only be accessed within member functions of that class.

3.10 Self Assessment Questions

Q.1. What is the use of a constructor function in a class? Give a suitable example of a constructor function in a class.

Q.2. Design a class having the constructor and destructor functions that should display the number of object being created or destroyed of this class type.

Q. 3. Write a C++ program, to find the factorial of a number using a constructor and a destructor (generating the message “you have done it”)

Q. 4. Define a class “string” with members to initialize and determine the length of the string. Overload the operators '+' and '+=' for the class “string”.

Unit - 4

Function in C++

Objective:

- Introduction to the concept of Functions
- Passing Information via Parameters
- Default Arguments
- Constant Arguments
- Function Overloading
- Inline Functions
- Recursive Functions

Structure:

4.1. Introduction to the Concept of Functions

Functions serve as the fundamental building blocks of C++ programs, encapsulating sets of declarations and statements to perform specific tasks.

4.1.1. Need for a Function

In large programs, a monolithic structure consisting of a single list of instructions can become challenging to comprehend. To address this, functions are employed. Functions offer several benefits:

1. Clear Objectives: Each function serves a clearly defined purpose, aiding in the organization and understanding of the program's logic.
2. Defined Interface: Functions possess a well-defined interface with other functions within the program, facilitating modular design and code reusability.
3. Reduction in Program Size: By breaking down the program into smaller, manageable functions, overall program size is reduced. Additionally, functions' code is stored in memory only once, even if executed multiple times, optimizing memory usage.

By leveraging functions, programmers can enhance the clarity, modularity, and efficiency of their C++ programs, leading to improved code maintenance and readability.

4.2. Declaration of Function and Definition

A function in C++ needs to be declared before it can be used in a program. The code for the function is contained in the function definition. The program's display_message() function declaration is located beneath the main() function. The following illustrates the general syntax of a C++ function definition

```
Type name_of_the_function (argument list)
{
//body of the function
}
```

Here, the type specifies the type of the value to be returned by the function. It may be any valid C++ data type. When no type is given, then the compiler returns an integer value from the function.

Name_of_the_function is a valid C++ identifier (no reserved word allowed) defined by the user and it can be used by other functions for calling this function.

Argument list is a comma separated list of variables of a function through which the function may receive data or send data when called from other function.

The above function add ( ) can also be coded with the help of arguments of parameters as shown below:

```
//function definition add()
void add(int a, int b) //variable names are must in definition

{
int sum; sum=a+b;
cout<<"\n The sum of two numbers is "<<sum<<endl;
}
```

4.3. Function Arguments

Arguments(s) of a function is (are) "data that function receives when called/invoked from another function".

4.3.1 Passing Arguments To A Function

Arguments(s) of a function is (are) the data that the function receives when called/invoked from another function.

It is not always necessary for a function to have arguments or parameters. The functions add ( ) and divide ( ) in program 6.3 did not contain any arguments. The following example illustrates the concept of passing arguments to function SUMFUN ( ):

```
// demonstration of passing arguments to a function #include<iostream.h>
 void main ()
  {
 float x,result; //local variables
 int N
```

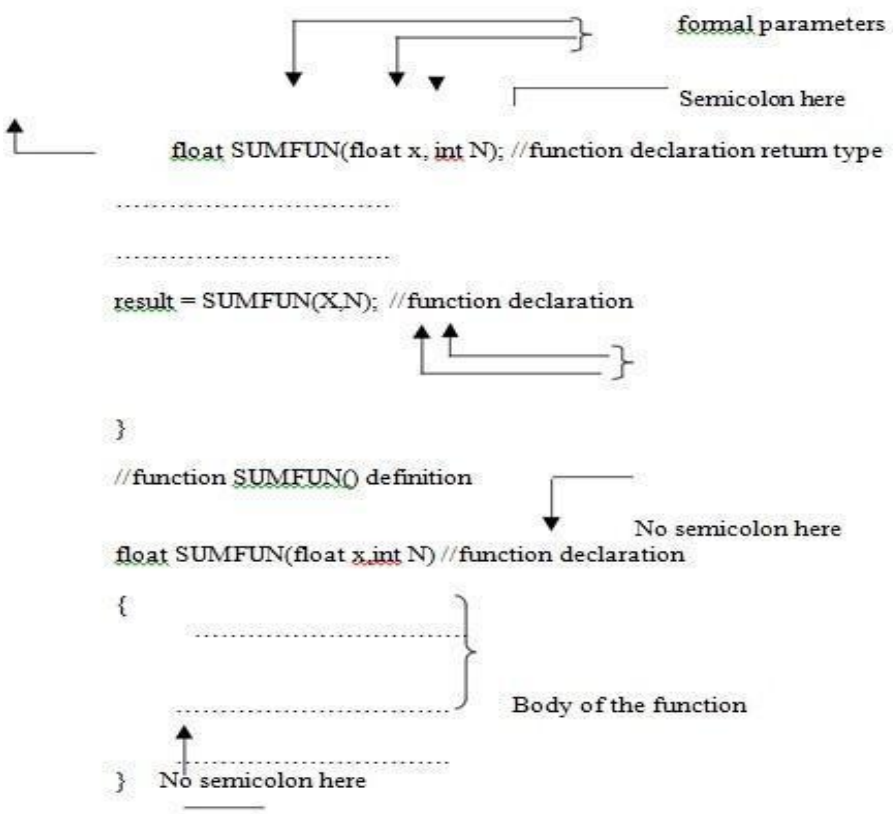


Fig 4.1 Argument Passing

4.3.1 Default Arguments

C++ allows a function to assign a parameter the default value in case no argument for that parameter is specified in the function call..

```
#include<iostream>
using namespace std;
int calc(int U) {
   if (U % 2 == 0)
      return U + 10;
   else
      return U + 2;
}

void pattern(char M, int B = 2) {
   for (int CNT = 0; CNT < B; CNT++)
      cout << calc(CNT) << M;
   cout << endl;
```

```
}
int main() {
    pattern('*');
    pattern('#', 4);
    pattern('@', 3);
    return 0;
}
```

4.3.2 Constant Arguments

In C++, functions may have constant arguments, which are treated as unmodifiable values within the function. To designate arguments as constant, the keyword const is used in the function prototype, as demonstrated below:

```
void max(const float x, const float y, const float z);
```

In this function prototype, the const qualifier indicates to the compiler that the arguments marked as const should not be altered by the function max(). This feature proves beneficial when employing call-by-reference methods for passing arguments.

Constant arguments play a crucial role in ensuring the integrity and immutability of data within functions, enhancing code reliability and maintainability.

4.4 Calling Functions

In C++ programs, functions with arguments can be invoked by:

(a) “Value”

(b) “Reference”

4.4.1 Call by “Value”

In this method, “values of actual parameters (appearing in the function call) are copied into the formal parameters”. Essentially, the function creates its own copy of argument values and operates on them. Below is an illustration of this concept:

```
// Calculation of compound interest using a function
#include<iostream>
#include<conio.h>
#include<math.h> // for pow() function
void calculate(float, float, float); // Function prototype
int main() {
```

```
    float principal, rate, time; // Local variables
    std::cout << "\nEnter the following values:\n";
    std::cout << "\nPrincipal: ";
    std::cin >> principal;
    std::cout << "\nRate of interest: ";
    std::cin >> rate;
    std::cout << "\nTime period (in years): ";
    std::cin >> time;
    calculate(principal, rate, time); // Function call
    getch();
    return 0;
}
// Function definition for calculate()
void calculate(float p, float r, float t) {
    float interest; // Local variable
    interest = p * (pow((1 + r / 100.0), t)) - p;
    std::cout << "\nCompound interest is : " << interest;
}
```

4.4.2 Call by Reference

A reference in C++ provides an alias, or an alternate name, for a variable. This means that the same variable's value can be accessed using two different names: the original name and the alias name.

In call by reference method, a reference to the actual arguments in the calling program is passed (only variables). Therefore, the called function does not create its own copy of the original values but works directly with the original values using different names. Any modification made to the original data within the called function is reflected back to the calling function.

This method is particularly useful when there is a need to alter the original variables in the calling function through the called function.

```
 // Swapping of two numbers using function call by reference
#include<iostream>
#include<conio.h>
int main() {
```

```
    clrscr();
    int num1, num2
    std::cout << "Enter two numbers: ";
    std::cin >> num1 >> num2;
    std::cout << "\nBefore swapping:\nNum1: " << num1;
    std::cout << "\nNum2: " << num2;
    swap(num1, num2); // Function call
    std::cout << "\n\nAfter swapping:\nNum1: " << num1;
    std::cout << "\nNum2: " << num2;
    getch();
    return 0;
}
// Function definition for swap()
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

4.5 INLINE FUNCTIONS.

Inline functions are designed to enhance program execution speed. When an inline function is called, its code is expanded directly at the location of the function call, rather than jumping to another location in memory for execution. This eliminates the overhead associated with function calls in regular functions, resulting in faster program execution.

The syntax for defining an inline function is as follows:

```
inline function_header {
    // Body of the function
}
```

For example:

```
 // Inline function definition for min()
inline void min(int x, int y) {
    cout << (x < y ? x : y);
}
int main() {
```

```
  int num1, num2;
    cout << "Enter two integers: ";
    cin >> num1 >> num2;
    min(num1, num2); // Function code inserted here
    //...
}
```

4.6 Scope Rules of Functions and Variables

The scope of an identifier is that part of the C++ program in which it is accessible. Generally, users understand that the name of an identifier must be unique. It does not mean that a name can't be reused. We can reuse the name in a program provided that there is some scope by which it can be distinguished between different cases or instances.

In C++ there are four kinds of scope as given below :

1. Local Scope
2. Function Scope
3. File Scope
4. Class Scope

Note that an inline function must be defined before it is invoked. In the example above, the min() function is declared as inline, so its code is inserted directly into the main() function at the point where it is called.

However, it's important to note that using inline functions has some limitations:

1. Inline functions are not suitable for large functions due to the memory penalty incurred by duplicating the function's code.
2. Inlining does not work for functions returning values and containing loops, switches, or goto statements.
3. Functions with static variables cannot be declared as inline.
4. Recursive functions cannot be declared as inline.

Despite these limitations, inline functions offer several benefits:

1. They are preferable to macros, providing type safety and other advantages.
2. Inline functions eliminate the overhead associated with function calls, leading to more efficient program execution.
3. They improve code readability by keeping the function definition close to its invocation.
4. Inline functions contribute to more efficient program execution.

4.6.1 Local Scope

In C++, a block is delineated by a pair of curly braces {}. Variables declared within the body of a block are termed local variables and are accessible only within that block. These variables come into existence when program control enters the block and cease to exist when control exits the closing brace.

It's important to note that local variables are available to all enclosed blocks within a block.

For example:

```
int x = 100;
{
 cout << x << endl;
  int x = 200;
    {
   cout << x << endl;
   int x = 300;
        {
  cout << x << endl;
       }
   cout << x << endl;
    }
   cout << x << endl;
}
```

4.6.2 Function Scope

Function scope refers to the labels declared within a function, meaning that a label can only be used within the function in which it is declared. Consequently, the same label names can be used in different functions without conflict.

For example:

```
// Function definition for add1()
void add1(int x, int y, int z) {
    int sum = 0;
    sum = x + y + z;
    cout << sum;
}
```

```
// Function definition for add2()
void add2(float x, float y, float z) {
   float sum = 0.0;
   sum = x + y + z;
   cout << sum;
}
```

4.6.3 File Scope

If the declaration of an identifier appears outside all functions, it becomes available to all functions in the program, and its scope extends to the entire file. This scope is termed as file scope.

For example:

```
int x;
void square(int n) {
   cout << n * n;
}
void main() {
   int num;
   cout << x << endl;
   cin >> num;
   square(num);
   // Additional code...
}
```

4.6.4 Class Scope

In C++, each class maintains its own associated scope. The members of a class have local scope within the class. If a variable name used by a class member is already defined with file scope, the variable will be hidden within the class. Member functions also have class scope.

Inline Functions

An inline function is expanded directly at the point where it is invoked.

Member Functions

Functions declared as private within a class can only be accessed by other functions within the same class.

Classes

When defining a class, you are specifying the attributes and behavior of a new type.

Objects

Declaring a variable of a class type creates an object. Multiple variables of the same class type can be created.

In summary, class scope encapsulates the members and member functions of a class, providing encapsulation and data hiding within the class structure. Inline functions offer efficiency by expanding directly where invoked, reducing function call overhead. Member functions, particularly private ones, ensure data integrity and security within the class. Classes and objects provide a structured way to organize and manipulate data, facilitating modular and reusable code design.

4.7 Summary

In this chapter, we explored the concept of functions in C++, including their declaration and definition. We also delved into the concept of classes, covering their declaration and definition. Additionally, we discussed methods for creating objects and accessing the data members of a class. Furthermore, we examined how to pass objects as arguments to functions using both call by value and call by reference techniques. Through these discussions, we gained a deeper understanding of how functions and classes operate in C++, paving the way for more advanced programming concepts.

4.8. Self Assessment Questions

1. Explain the concept of a function in C++. How do you define a function, and what are its essential components?
2. Discuss how argument data types are specified for functions in C++, using suitable examples to illustrate your explanation.
3. Enumerate the different types of functions available in C++, and elaborate on each type with relevant details.
4. Define recursion in programming. What precautions should be taken while implementing a recursive function in C++?
5. Define an inline function and describe the scenarios in which you would choose to make a function inline, providing reasons for your choices.
6. Define a class in C++ and explain how objects of a class are instantiated. Provide an example to illustrate your explanation.

7. Explain the significance of the scope resolution operator (::) in C++ programming, and demonstrate its usage with examples.
8. Define data members, member functions, private members, and public members within the context of classes in C++. Provide examples to elucidate each concept.
9. Write the class definition for a class named Student, adhering to the provided specifications. Explain each component of the class definition in detail.
10. Define a string data type in C++ with the specified functionality, including constructors, copy constructor, destructor, and overloaded operators. Provide a comprehensive explanation for each component.

Unit 5

Classes and Objects

Objective:

- Introduction Class,
- Understand the Member Functions,
- Making an Outside Function Inline,
- Nesting Of Member Functions ,
- Private Member Function,
- Arrays within a Class,
- Memory Allocation for Objects,
- Arrays of Objects,
- Objects as Function Arguments, Returning Objects,
- Const Member Function,
- Static Class Members,
- Pointer to Members,
- Local classes,
- Friend Functions,
- Unions and classes,
- Object Composition and Delegation

Structure :

5.1. Introduction of a Class,
5.2. Member Functions,
5.3. Declaration of Objects as Instance of a Class
5.4. Friend Class
5.5. Drawbacks of Preprocessors/Macros in C++
5.6. Inline Functions in C++
5.7. Important points about Inline Functions
5.8. Getter and Setter Functions in C++,
5.9. Limitations of Inline Functions
5.10. Understanding Forward References in C++
5.11. Function Overloading in C++
5.12. Summary.
5.13.Self Assessment Questions

5.1 Introduction of a Class

In C++, a class is a grouping of related variables and functions. It creates a data type that can be utilized to create this kind of object. Classes are representations of actual entities in the real world with corresponding operations (behavior) and data type features (characteristics). Below is an example of a class definition's syntax:

```
  Class name_of _class
{
: variable declaration; // data member
Function declaration; // Member Function (Method)
protected: Variable declaration;
Function declaration;
public : variable declaration;
{
Function declaration;
};
```

In C++, the keyword **class** signifies the usage of a new data type, followed by the class name. The body of the class incorporates two keywords:

1. private
2. public

These keywords, **private** and **public**, are known as access specifiers. The concept of data hiding in C++ is realized through the **private** keyword. Private data and functions can only be accessed from within the class itself. Conversely, public data and functions are accessible from outside the class as well.

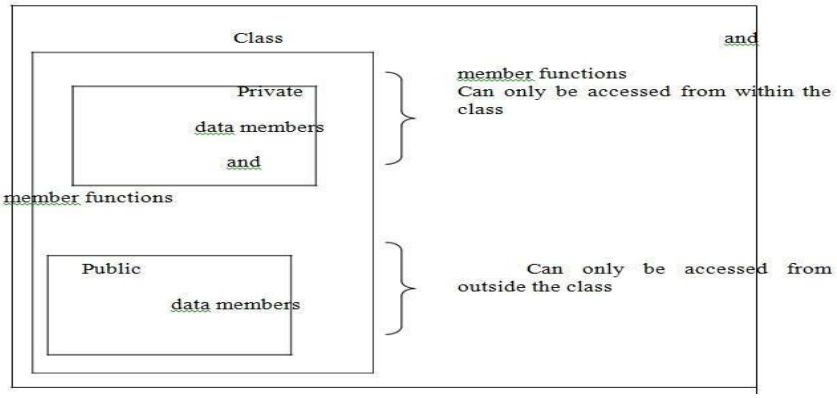


Figure 5.1

Data hiding in the context of object-oriented programming does not refer to the security technique used to protect computer databases. Instead, it pertains to a programming practice aimed at safeguarding data within a class from unintended manipulation.

Data declared under the private section of a class remains hidden and secure from accidental manipulation. While users can access private data, they cannot do so inadvertently.

Typically, functions that operate on the data within a class are declared as public. This allows them to be accessed from outside the class. However, it's important to note that this is not a strict rule that must always be followed.

By encapsulating data within a class and controlling its access through public and private members, developers can ensure data integrity and prevent unintended side effects in their programs.

5.2 Member Function Definition

In C++, the specification of a class can be divided into two parts:

1. **Class definition**: This part encompasses the declaration of both data members and member functions.
2. **Class method definitions**: This part outlines how certain class member functions are implemented.

We have previously examined the syntax and an example of class definition. In C++, member functions can be implemented in two ways:

(a) **Inside class definition**: The member function is defined within the class body itself.

(b) **Outside class definition using the scope resolution operator (::)**: The member function is defined outside the class body, and the scope resolution operator (::) is used to specify that it belongs to a particular class.

While the code of the function remains the same in both cases, the function header differs:

- When the function is defined inside the class, it is implicitly assumed to belong to that class.
- When the function is defined outside the class, the class name followed by the scope resolution operator (::) is used to specify the class to which the function belongs.

This flexibility in defining member functions allows for better organization and management of class implementations in C++.

5.2.1 Inside Class Definition:

When a member function is defined inside a class, there's no need to include a membership label along with the function name. Typically, only small functions are defined within the class, and these are referred to as inline functions.

In the case of inline functions, the compiler embeds the code of the function body directly at the place where it is invoked, resulting in faster program execution. However, this approach may incur a memory penalty.

This strategy of defining functions inline within the class is beneficial for enhancing code readability and maintaining a compact codebase.

5.2.2 Outside Class Definition:

In the scenario where a member function is defined outside a class, the function's full name, known as the qualified name, is written as follows: **Name_of_the_class::function_name.**

The syntax for defining a member function outside the class definition is as follows:

```
return_type name_of_the_class::function_name(argument list)
{
   // body of function
}
```

Here, the scope resolution operator (::) is utilized to specify the class to which the function belongs. The scope resolution operator aids in delineating the member function outside the class.

This methodology enables a clear separation and organization of class member functions, enhancing code readability and maintainability.

5.3. Declaration of Objects as Instances of A Class:

Following the class definition comes the declaration of the class's objects. It is important to keep in mind that a class description describes a class's properties rather than any objects of that kind. We require variables of the class type in order to use the defined class. As an illustration,

The largest ob1, ob2; //object declaration will produce the largest class type objects, ob1 and ob2. As was previously established, a class's variables in C++ are referred to as objects. These are declared in the same way as basic data types, or as a simple variable.

When a class is defined in C++, all of its member functions are generated and saved in memory, which is accessible to all of the class's associated objects.

Every object has a different amount of memory allocated to it for each of its data members. Different values are stored in member variables for various class objects.

5.3.1. Accessing Members From Object(S)

After defining a class and creating a class variable, i.e., an object, we can access the data members and member functions of the class. Since the data members and member functions are part of the class, they must be accessed using the variables we created. For example:

```
class Student {
private:
   char reg_no[10];
   char name[30];
   int age;
   char address[25];
public:
   void init_data() {
      // body of function
   }
   void display_data() {
      // body of function
   }
};
Student ob; // class variable (object) created
ob.init_data(); // accessing the member function
ob.display_data(); // accessing the member function
```

Here, the data members can be accessed within the member functions as they have private scope, and the member functions can be accessed outside the class, i.e., before or after the **main()** function.

5.3.2 Static Class Members:

In C++, both data members and member functions of a class may be qualified as static. We can have static data members and static member functions in a class.

Static Data Member: It is typically used to store a value common to the entire class. The static data member differs from an ordinary data member in the following ways:

1. Only a single copy of the static data member is used by all the objects.
2. It can be accessed within the class, but its lifetime spans the entire program.

To make a data member static, we need to:

- Declare it within the class.
- Define it outside the class. For example:

```
class Student {
   static int count; // declaration within class
};
```

The static data member is then defined outside the class:

```
int Student::count; // definition outside class
```

The definition outside the class is necessary. We can also initialize the static data member at the time of its definition, such as:

```
int Student::count = 0;
```

If we define three objects as:

```
Student obj1, obj2, obj3;
```

Static Member Function: A static member function can only access the static members of a class. This is done by prefixing the keyword static before the name of the function while declaring it. For example:

```
class Student {
   static int count;
public:
   static void showcount(); // static member function
};
```

Here, the keyword **static** is placed before the name of the function **showcount().** In C++, a static member function differs from other member functions in the following ways:

1. Only static members (functions or variables) of the same class can be accessed by a static member function.
2. It is called using the name of the class rather than an object. For example:

```
Student::showcount();
```

This usage allows for better management and organization of class functionality in C++.

5.4. Friend Classes

In C++, a class can be designated as a friend to another class, granting access to its private members. For instance:

```
class TWO; // forward declaration of the class TWO
class ONE {
   // ...
public:
   // ...
   friend class TWO; // class TWO declared as friend of class ONE
};
```

Here, from class TWO, all members of class ONE can be accessed. Friend functions in C++ are not class member functions; instead, they provide private access to non-class functions.

For example:

```
class WithFriend {
   int i;
public:
   friend void fun(); // global function as friend
};
void fun() {
   WithFriend wf;
   wf.i = 10; // access to private data member
   cout << wf.i;
}
int main() {
   fun(); // Can be called directly
}
```

Thus, friend functions can access private data members by creating an object of the class. Similarly, functions from another class can be designated as friends, or an entire class can be made a friend class.

```
class Other {
   void fun();
};
class WithFriend {
private:
```

```
    int i;
public:
    void getdata(); // Member function of class WithFriend
    // making function of class Other as friend here
    friend void Other::fun();
    // making the complete class as friend
    friend class Other;
};
```

When a class is made a friend, all its member functions automatically become friend functions. This feature in C++, however, compromises the concept of encapsulation and is one reason why C++ is not considered a purely object-oriented language.

Additionally, in C++, all member functions defined inside the class definition are by default declared as inline functions. It's worth noting the similarity between inline functions and preprocessors/macros from C language, although inline functions offer advantages over macros.

5.5. Drawbacks of Preprocessors/Macros in C++:

In macros, we define certain variables with their values at the beginning of the program, and everywhere inside the program where we use that variable, it's replaced by its value during compilation.

Problem with spacing: Let's illustrate this problem using an example:

#define G(y) (y+1)

Here, we've defined a macro named G(y), which is to be replaced by its value (y+1) during compilation. However, when we call G(1), the preprocessor expands it as:

(y) (y+1) (1)

This unexpected expansion occurs due to spacing in the macro definition. Consequently, macros are unsuitable for large functions with multiple expressions, prompting the introduction of inline functions in C++.

Complex Argument Problem: In some cases, macro expressions work fine for certain arguments, but problems arise with complex arguments. For instance:

#define MAX(x,y) x>y?1:0

Now, if we use the expression:

if(MAX(a&0x0f, 0x0f))

The macro expands to:

if(a&0x0f > 0x0f ? 1:0)

Due to the precedence of operators, the macro evaluation may yield unexpected results. This problem can be mitigated by using parentheses, but issues persist with larger expressions.

No way to access Private Members of Class: With macros in C++, you cannot access private variables. Consequently, you'll need to make those members public, thereby exposing the implementation.

```
class Y {
   int x;
public:
   #define VAL(Y::x) // Error
};
```

This attempt to access a private member with a macro results in an error, highlighting the limitation of macros in accessing private class members.

5.6. "Inline Functions in C++"

Inline functions in C++ are actual functions that are copied everywhere during compilation, similar to preprocessor macros. This copying process reduces the overhead of function calling. By default, all functions defined inside a class definition are inline. Additionally, you can also make any non-class function inline by using the **inline** keyword with them.

For an inline function, the declaration and definition must be done together. For example:

```
inline void fun(int a) {
 return a++;
}
```

Here, the **inline** keyword indicates that the function **fun** should be treated as an inline function. This function will be copied wherever it is called in the program, helping to reduce the overhead associated with function calls.

5.7. "Important points about Inline Functions:"

1. It's advisable to keep inline functions small, as smaller inline functions tend to have better efficiency.
2. While inline functions do enhance efficiency, it's important not to make all functions inline. Making large functions inline may result in code bloat, potentially affecting the program's speed.
3. Therefore, it's recommended to define large functions outside the class definition

using the scope resolution operator (::). By defining such functions outside the class, they won't automatically become inline.

4. Inline functions are stored in the Symbol Table by the compiler, and all calls to such functions are resolved at compile time. This compile-time resolution helps optimize program performance.

5.8. Getter and Setter Functions in C++

This was previously covered in the section on accessing private data variables within a class. We utilize

Auto

```
“// by default private int
price;
public:
// getter function for variable price int
getPrice()
{
return price;

}
// setter function for variable price void
setPrice(int x)
{
i=x;
}
};ss functions, which are inline to do so.
```

In this case, the inline procedures getPrice() and setPrice() are designed to access the private data members of the class Auto. In this instance, the function setPrice() is a Setter or Mutator function, and the function getPrice() is referred to as a Getter or Accessor function. Additionally, accessor and mutator functions may overlap. In the following article, we will examine overloading functions.

5.9. Limitations of Inline Functions:

1. Performance is adversely affected by large inline routines, which result in cache misses.

2. The function body is copied across the code during compilation, which adds compilation cost that is insignificant for small applications but significant for big code bases.
3. Moreover, the compiler is unable to execute code on functions that require the address of the function within the program. Because the compiler must allot storage to a function in order to provide it an address. However, inline functions are stored in the Symbol table instead of being stored.

5.10.Understanding Forward References in C++

All inline functions within a class are evaluated by the compiler at the end of the class declaration. Consider the following example:

```
class ForwardReference {
   int i;
public:
   // call to undeclared function int f()
   {
      return g() + 10;
  }
   int g() {
      return i;
   }
};
int main() {
   ForwardReference fr;
   fr.f();
}
```

You might expect this code to result in a compile-time error due to the call to an undeclared function **f()**. However, in this case, it will work because no inline function in a class is evaluated until the closing braces of the class declaration.

5.11. Function Overloading in C++

A class is considered to be overloaded if it contains numerous functions with the same name but different parameters. You can perform the same or various functions in the same class using the same name thanks to function overloading.

Function overloading is typically employed to improve the program's readability. You can overload the function if all you need to do is execute one action with various numbers or kinds of parameters.

- **Different ways to overload a function in C++ include:**
  1. By changing the number of arguments.
  2. By having different types of arguments.

These methods allow multiple functions with the same name to exist within the same scope, as long as they have different parameter lists. This enables greater flexibility and versatility in function usage within a program.

- **Function Overloading: Different Number of Arguments**

  I We define two functions with the same names but differing numbers of the same type of parameters in this type of function overloading. For instance, we have created two sum() functions in the program listed below to yield the sum of two and three integers.

```
   // first definition int
sum (int x, int y)
{
cout << x+y;
}
// second overloaded defintion int
sum(int x, int y, int z)
{
 cout << x+y+z;
 }
```

Here the sum() function is said to be overloaded, as it has two definitions, one which accepts two arguments and another which accepts three arguments. Which sum() function will be called, depends on the number of arguments.

```
 int main()
{
 // sum() with 2 parameter will be called sum
 (10, 20);
 //sum() with 3 parameter will be called
 sum(10, 20, 30);
```

} 30

60

- **Function Overloading: Different Type of Arguments**

A parameter is considered to be default when its value is specified when the function is declared. In this instance, the function will use the provided default value even if we call it without passing in a value for that parameter.

```
sum(int x, int y=0)
{
cout << x+y;
}
```

Here we have provided a default value for y, during function definition.

```
int main()
{
sum(10);
sum(10,0);
sum(10,10);
}
```

10 10 20

First two function calls will produce the exact same value for the third function call, y will take 10 as value and output will become 20. By setting the default argument, we are also overloading the function. Default arguments also allow you to use the same function in different situations just like function overloading.

- **Rules for using Default Arguments**

1. Only the last argument must be given a default value. You cannot have a default argument followed by a non-default argument.
2. sum(int x, int y);
3. sum(int x, int y = 0);
4. sum(int x = 0, int y); // This is Incorrect
5. If you default an argument, then you will have to default all the subsequent arguments after that.
6. sum(int x, int y = 0);
7. sum(int x, int y = 0, int z); // This is incorrect
8. sum(int x, int y = 10, int z = 10); // Correct

9. You can assign any compatible value as a default value to an argument.

- **Function with Placeholder Arguments**

  When arguments in a function are declared without any identifier they are called placeholder arguments. void sum (int, int);

  Such arguments can also be used with default arguments. void sum (int, int=0);

5.12. Summary

You have learned about base classes and derived classes in this chapter. A class that contains a member of another class is called derived. Another name for this idea is inheritance. Multiple Inheritance refers to the situation where a derived class has more than one direct base class. Three different kinds of inheritance existed. Additionally, classes can be declared as members of other classes. Additionally, we discussed the idea of polymorphism.

.

5.13. Self Assessment Questions:

1. What are Functions? Explain different Functions in C++.
2. What is 'Calling a Function'? Explain different types of calling a function.
3. What are Classes and Objects? Explain the procedure for the same.
4. What is a Constructor? What is a Destructor? Illustrate with example Constructors and destructors.
5. Illustrate with example Access Control in C++.
6. Illustrate with example Calling Class Member Function in C++.
7. Explain following types of Class Member Functions in C++:
   - Simple functions
   - Static functions
   - Const functions
   - Inline functions
   - Friend functions
8. Drawbacks of Preprocessors/Macros in C++.
9. What are Inline Functions? Explain the Important points about Inline Functions.
10. What is understood by Understanding Forward References in C++?
11. Explain with illustration the Function Overloading in C++.

12. Provide Examples of Various Methods for Overloading a Function

    i. By altering the quantity of arguments.

    ii. By engaging in various forms of debate .

13. Explain the types of Constructors in C++

    i. Default Constructor

    ii. Parameterized Constructor

    iii. Copy Constructor

Unit 6

Constructor and Destructor

Objective:

- Understand the concept of Constructor
- Multiple Constructors in a class,
- Constructor with Default Arguments,
- Dynamic Initialization of Objects,
- Const Object,
- Destructor

Structure:

6.1 Introduction to Constructor

A constructor (having the same name as that of the class) is a member function which is automatically used to initialize the objects of the class type with legal initial values. Destructors are the functions that are complimentary to constructors. These are used to de-initialize objects when they are destroyed. A destructor is called when an object of the class goes out of scope, or when the memory space used by it is de-allocated with the help of delete operator.

Operator overloading is one of the most exciting features of C++. It is helpful in enhancing the power of extensibility of C++ language. Operator overloading redefines the C++ language. User defined data types are made to behave like built-in data types in C++. Operators +, *. <=, += etc. can be given additional meanings when applied on user defined data types using operator overloading. The mechanism of providing such an additional meaning to an operator is known as operator overloading in C++.

6.2 Declaration and Definition of a Constructor

It is defined like other member functions of the class, i.e., either inside the class definition or outside the class definition.

For example, the following program illustrates the concept of a constructor :

```
 //To demonstrate a
constructor #include
<iostram.h> #include
<conio.h>
Class rectangle
{ private :
float length, breadth; public:
rectangle ()//constructor definition
{
//displayed whenever an object is created cout<<”I
 am in the constructor”; length-10.0; breadth=20.5;
}
float area()
{
return (length*breadth);
}
```

```
};
void main()
 clrscr();
 rectangle rect; //object declared
 cout<<"\nThe area of the rectangle with default parametis:"<<rect.area()<<"sq.units\n";
 getch();
 }
```

6.3 Type Of Constructor

There are different types of constructors in C++.

6.3.1 Overloaded Constructors

Besides initializing member data, constructors are similar to other functions, including their ability to be overloaded. Overloading constructors is a common practice. For instance, consider the following program with overloaded constructors for the "Figure" class:

```
  //Illustration of overloaded constructors
 //construct a class for storage of dimensions of circles.
 //triangle and rectangle and calculate their area
#include<iostream.h>
#include<conio.h>
#include<math.h>
#include<string.h> //for
strcpy() Class figure
 {
 Private:
 Float radius, side1,side2,side3; //data members Char
 shape[10]; Public:
 figure(float r) //constructor for circle
{
 radius=r;
 strcpy (shape, "circle");
 }
 figure (float s1,float s2) //constructor for
 rectangle strcpy
```

```
    {
    Side1=s1;
Side2=s2;
"Side3=radius=0.0; //has no significance in rectangle
strcpy(shape,"rectangle");" }
    Figure (float s1, floats2, float s3) //constructor for triangle
    {
        side1=s1; side2=s2; side3=s3; radius=0.0;
        strcpy(shape,"triangle");
    }
    void area() //calculate area
    {
        float ar,s; if(radius==0.0)
        if (side3==0.0)
        {
            ar=side1*side2; else
            ar=6.14*radius*radius;
            cout<<"\n\nArea of the "<<shape<<"is :"<<ar<<"sq.units\n";
        }
    };
    Void main()
    {
        Clrscr();
        Figure circle(10.0); //object initialized using constructor Figure
        rectangle(15.0,20.6);//object initialized using constructor
        Figure Triangle(6.0, 4.0, 5.0); //object initialized
        using constructor Rectangle.area();
        Triangle.area(); Getch();//freeze the monitor
    }
```

6.3.2 Copy Constructor

It is of the form "classname (classname &)" and used for "initialization of an object form another object of same type". For example.

```
    Class fun
```

```
{
Float x,y;
Public:
Fun (floata,float b)//constructor
{x = a; y = b;
}
Fun (fun &f) //copy constructor {cout<<"\ncopy constructor at work\n";
X = f.x; Y = f.y;
 }
Void display (void)
{
{
Cout<<""<<y<<end1;
}
};
```

Here we have "two constructors", "one copy constructor" for copying data value of a fun object to another and other "one a parameterized constructor" for assignment of initial values given.

6.3.3 Constructors and Primitive Types

In C++, like derived type, i.e. class, primitive types (fundamental types) also have their constructors. "Default constructor is used when no values are given but when we given initial values, the initialization take place for newly created instance"

example,

"float x,y; //default constructor used

int a(10), b(20); //a,b initialized with values 10 and 20 float i(2.5), j(7.8); //I,j, initialized with valurs 2.5 and 7.8"

6.4 Constructor with "Default Arguments"

In C++, we can define constructors with "default arguments". For example,

```
Class add
{
Private:
Int num1, num2,num3; Public:
```

```
Add(int=0,int=0); //Default argument constructor //to reduce the number of constructors Void enter (int,int);
Void sum();
Void display();
};
//Default constructor definition add::add(int n1, int n2)
{
num1=n1; num2=n2; num3=n0;
}
Void add ::sum()
{
Num3=num1+num2;
 }
 Void add::display ()
{
Cout<<"\nThe sum of two numbers is "<<num3<<end1;
}
```

"Now using the above code objects of type add can be created with no initial values, one initial values or two initial values". For Example,

```
 Add obj1, obj2(5), obj3(10,20);
```

Here, obj1 will have values of data members num1=0, num2=0 and num3=0

Obj2 will have values of data members num1=5, num2=0 and num3=0 Obj3 will have values of data members num1=10, num2=20 and num3=0

If two constructors for the above class add are Add::add() {} //default constructor and add::add(int=0);//default argument constructor

Then the default argument constructor can be invoked with either two or one or no parameter(s). Without argument, it is treated as a default constructor-using these two forms together causes ambiguity. For example,

The declaration add obj;

is ambiguous i.e., which one constructor to invoke i.e., add :: add()

or add :: add(int=0,int=0)

so be careful in such cases and avoid such mistakes".

Special Characteristics Of Constructors

1. **Automatic Invocation**: Constructors are automatically called when objects are created.
2. **Initialization**: All objects of a class with a constructor are initialized before use.
3. **Public Declaration**: Constructors should be declared in the public section to be accessible to all functions.
4. **No Return Type**: Constructors cannot have a return type, not even void.
5. **Non-Inheritable**: Constructors cannot be inherited, although a derived class can call the base class constructor.
6. **Non-Static**: Constructors cannot be static.
7. **Compiler-Generated Constructors**: The compiler generates default and copy constructors as needed, and these generated constructors are public.
8. **Default Arguments**: Constructors can have default arguments, similar to other C++ functions.
9. **Member Function Calls**: Constructors can call member functions of their class.
10. **Union Limitation**: An object of a class with a constructor cannot be used as a member of a union.
11. **Creating New Objects**: Constructors can be used to create new objects of their class type using the syntax `ClassName(expression_list)`. For example:

    ```cpp
    Employee obj3 = obj2; // See program 10.5
    // Or even
    Employee obj3 = Employee(1002, 35000); // Explicit call
    ```

12. **Memory Allocation**: Constructors implicitly call the memory allocation (`new`) and deallocation (`delete`) operators.
13. **Non-Virtual**: Constructors cannot be virtual.

6.5 Destructor Declaration

"The syntax for declaring a destructor is :

-name_of_the_class()

{

}

So the name of the class and destructor is same but it is prefixed with a ~

(tilde). It does not take any parameter nor does it return any value. Overloading a destructor is not possible and can be explicitly invoked. In other words, a class can have only one destructor. A destructor can be defined outside the class. The following program illustrates this concept :

```
  //Illustration of the working of Destructor function
 #include<iostream.h>
#include<conio.h> class
 add
 {
 private :
 int num1,num2,num3; public :
 add(int=0, int=0); //default argument constructor
//to reduce the number of constructors"
 void sum();
 void display();
 ~ add(void); //Destructor
 };
 //Destructor definition ~add()
 Add:: ~add(void) //destructor called automatically at end of program
  {
 Num1=num2=num3=0;
 Cout<<"\nAfter the final execution, me, the object has entered in the"
 <<"\ndestructor to destroy myself\n"; }
 //Constructor definition add()
Add::add(int n1,int n2)
 {num1=n1; num2=n2; num3=0;
}
//function definition sum ()
Void add::sum()
 {
num3=num1+num2;
}
//function definition display () Void add::display ()
        {
```

```
        Cout<<"\nThe sum of two numbers is "<<num3<<end1;
        }
void main()
        {
        Add obj1,obj2(5),obj3(10,20): //objects created and initialized clrscr();
        Obj1.sum(); //function call
        Obj2.sum();
        Obj6.sum();
        cout<<"\nUsing obj1 \n";
        obj1.display(); //function call
        cout<<"\nUsing obj2 \n"; obj2.display();
        cout<<"\nUsing obj3 \n"; obj6.display();
        }
```

Special Characteristics of Destructors

Some of the characteristics associated with destructors are :

Destructors in programming exhibit several distinct characteristics:

1. **Automatic Invocation**: Destructors are called "automatically when objects are destroyed".
2. Access Rules: Destructor functions adhere to the usual access rules applicable to other member functions.
3. De-initialization: They are responsible for de-initializing each object before it goes out of scope.
4. No Arguments or Return Type: Destructors cannot have arguments or a return type, not even void.
5. Non-Inheritable: Destructors cannot be inherited by derived classes.
6. Static Destructors: Static destructors are not permitted.
7. Destructor Address: The address of a destructor cannot be taken.
8. Member Function Calls: A destructor can call member functions of its class.
9. Union Membership: An object with a destructor cannot be a member of a union.

These characteristics define the behavior and constraints of destructors in object-oriented programming.

6.6 Summary

In this chapter, we covered the concepts and types of constructors and destructors, as well as the operators that can be overloaded. Despite overloading operator functions, the precedence of operators remains unchanged. The '++' and '--' operators can be implemented as either postfix or prefix, and we provided examples of separate functions for each application. Additionally, it is generally accepted that the private data of a class should only be accessed within the member functions of that class .

6.7 Self Assessment Questions

1. What is a constructor in object-oriented programming?
2. How does a parameterized constructor differ from a default constructor?
3. What is constructor overloading? Provide an example.
4. Explain the role of a copy constructor. When is it called?
5. Can a constructor in C++ return a value? Why or why not?
6. What is a destructor in object-oriented programming?
7. What happens if a destructor is not explicitly defined in a class?
8. How do destructors ensure proper resource management in C++?
9. Can a destructor be overloaded? Explain why or why not.
10. What is the significance of the 'virtual' keyword in the context of destructors?

Unit : 7

Operator Overloading and Type Conversion

Objective:

- Introduction
- Concept of Overloading
- Unary Operators in C++,
- Overloading Binary Operators in C++,
- Limitations of Operator Overloading in C++
- This pointer concept,
- Overloading<<and>>Operators,
- Manipulation of String,
- Types Conversion

Structure:

7.1. Introduction,
7.2. Overloading Unary Operators,
7.3. Overloading Binary Operators,
7.4. Limitations of Operator Overloading,
7.5. This pointer,
7.6. Overloading<<and>>Operators,
7.7. Manipulation of String,
7.8. Types Conversion
7.9. Summary
7.10. Self Assessment Questions

7.1 Operator Overloading

Operator overloading stands as a pivotal concept within C++ programming, constituting a form of polymorphism wherein an operator assumes a user-defined significance. By overloading operators, users can imbue them with custom functionality tailored to specific data types. This enables the execution of operations on user-defined data types in a manner akin to built-in types. For instance, the '+' operator might be overloaded to execute addition operations on diverse data types such as integers or strings for concatenation purposes.

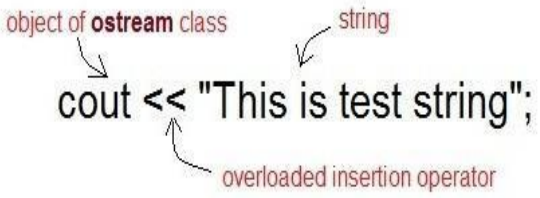


Fig 7.1 Operator Overloading Syntax

Operator overloading in C++ offers flexibility by allowing the definition of custom behaviors for operators. It can be achieved through various function implementations:

1. **Member Function:** Operator overloading functions can be defined as member functions of a class. This approach is suitable when the left operand is an object of that class.
2. **Non-Member Function:** When the left operand of an operator belongs to a different type than the class, operator overloading functions must be defined as non-member functions.
3. **Friend Function:** Operator overloading functions can also be declared as friend functions of a class. This is necessary when the function needs access to the private and protected members of the class.

7.2 Implementing Operator Overloading in C++

While implementing operator overloading in C++, it's crucial to adhere to certain restrictions:

1. **Precedence and Associativity:** Operators cannot change their precedence or associativity through overloading.
2. **Arity of Operators:** The number of operands an operator takes (unary, binary, etc.) cannot be altered.
3. **Creation of New Operators:** Operator overloading is limited to existing operators; it does not allow the creation of new operators.
4. **Redefinition of Procedures:** Operators cannot redefine the fundamental procedures

of operations. For instance, the meaning of integer addition cannot be changed.

7.3 Operator Overloading Examples in C++: Operator overloading enables the customization of almost all operators. Here's an example demonstrating the overloading of the addition operator to add two Time class objects:

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

class Time {
   int h, m, s;
public:
   Time() {
      h = 0, m = 0; s = 0;
   }
   void setTime();
   void show() {
      std::cout << h << ":" << m << ":" << s;
   }
   Time operator+(Time);
};
Time Time::operator+(Time t1) {
   Time t;
   int a, b;
   a = s + t1.s;
   t.s = a % 60;
   b = (a / 60) + m + t1.m;
   t.m = b % 60;
   t.h = (b / 60) + h + t1.h;
   t.h = t.h % 12;
   return t;
}
void time::setTime() {
   std::cout << "\nEnter the hour(0-11) ";
   std::cin >> h;
```

```
    std::cout << "\nEnter the minute(0-59) ";
    std::cin >> m;
    std::cout << "\nEnter the second(0-59) ";
    std::cin >> s;
}

int main() {
    Time t1, t2, t3;
    std::cout << "\nEnter the first time ";
    t1.setTime();
    std::cout << "\nEnter the second time ";
    t2.setTime();
    t3 = t1 + t2;
    std::cout << "\nFirst time ";
    t1.show();
    std::cout << "\nSecond time ";
    t2.show();
    std::cout << "\nSum of times ";
    t3.show();
    getch();
    return 0;
}
```

7.4 Overloading I/O operator in C++

Overloading input/output (I/O) operators in C++ offers customization for printing and reading values of user-defined data types. There are several scenarios where overloading these operators proves beneficial:

1. Customized Output: Overloading the output operator '<<' allows printing values of user-defined data types.
2. Customized Input: Overloading the input operator '>>' allows inputting values for user-defined data types.

When overloading these operators, the left operands will be of types ostream& and istream&. Since the left operand is not an object of the class, the overloading functions must be non-member functions and should be declared as friend functions to access

private data members.

The '<<' operator is already overloaded by default with ostream class objects like cout, enabling the printing of primitive data types to the screen. Similarly, we can overload '<<' in our class to print user-defined data types to the screen, making the usage more intuitive.

7.5 Overloading Relational Operators in C++

Overloading relational operators such as ==, !=, >=, <=, etc., allows for comparison between two objects of any class. Below is a quick example demonstrating the overloading of the == operator in the Time class to directly compare two Time class objects.

```
 class Time {
   int hr, min, sec;
public:
   // default constructor
   Time() {
      hr = 0, min = 0; sec = 0;
   }
   // overloaded constructor
   Time(int h, int m, int s) {
      hr = h, min = m; sec = s;
   }
   // overloading '==' operator
   friend bool operator==(Time &t1, Time &t2);
};

/*
Defining the overloading operator function
Here we are simply comparing the hour, minute, and second values of two different Time objects to compare their values
*/
bool operator== (Time &t1, Time &t2) {
   return ( t1.hr == t2.hr && t1.min == t2.min && t1.sec == t2.sec );
}
```

```
void main() {
   Time t1(3,15,45);
   Time t2(4,15,45);
   if(t1 == t2) {
      cout << "Both the time values are equal";
   }
   else {
      cout << "Both the time values are not equal";
   }
}
```

7.6. Copy Constructor vs. Assignment Operator (=)

The assignment operator is utilized to copy values from one object to another existing object. For instance:

```
Time tm(3,15,45); // tm object created and initialized
Time t1;          // t1 object created
t1 = tm;          // initializing t1 using ™
```

In the case of a copy constructor, we provide the object to be copied as an argument to the constructor. Additionally, we first need to define a copy constructor in our class.

For defining an additional task to an operator, we must specify what it means in relation to the class to which it applies. The operator function facilitates this task.

The syntax for declaring an operator function is as follows:

```
return_type operator Operator_name
```

A binary operator can be defined either as a member function taking one argument or a global function taking one argument. For a binary operator X, a X b can be interpreted as either an operator X (b) or operator X (a, b).

An operator function should be either a member or take at least one class object argument. Some examples of declarations of operator functions are given below:

7.7. Operator Overloading using Friend in Class Time

```
class Time {
   int r;
   int i;
public:
```

```
    friend Time operator+(const Time &x, const Time &y);
    // Constructor
    Time() {
        r = i = 0;
    }
    // Parameterized constructor
    Time(int x, int y) {
        r = x;
        i = y;
    }
};
// Overloading the '+' operator using friend function
Time operator+(const Time &x, const Time &y) {
    Time z;
    z.r = x.r + y.r;
    z.i = x.i + y.i;
    return z;
}
int main() {
    Time x, y, z;
    x = Time(5, 6);
    y = Time(7, 8);
    z = Time(9, 10);
    z = x + y; // addition using friend function '+'
}
```

7.8. Operator Overloading using Member Function: Class ABC

```
 class ABC {
    char *str;
    int len; // Present length of the string
    int max_length; // Maximum space allocated to string
public:
    // Constructors and Destructor
    ABC(); // Black string of length 0 with a maximum allowed length of size 10
```

```
    ABC(const ABC &s); // Copy constructor
    ~ABC() { delete[] str; }
    // Overloaded operators
    int operator==(const ABC &s) const; // Check for equality
    ABC &operator=(const ABC &s); // Overloaded assignment operator
    friend ABC operator+(const ABC &s1, const ABC &s2); // String concatenation
    // Member function declarations
    friend std::ostream &operator<<(std::ostream &s, const ABC &x); // Overloading the << operator
};
// Default constructor
ABC::ABC() {
    max_length = 10;
    str = new char[max_length];
    len = 0;
    str[0] = '\0';
}
// Copy constructor
ABC::ABC(const ABC &s) {
    len = s.len;
    max_length = s.max_length;
    str = new char[max_length];
    strcpy(str, s.str); // Physical copying to the new location
}
// Overloaded assignment operator
ABC &ABC::operator=(const ABC &s) {
    if (this != &s) { // Check for self-assignment
        len = s.len;
        max_length = s.max_length;
        delete[] str; // Deallocate old memory
        str = new char[max_length]; // Allocate new memory
        strcpy(str, s.str); // Copy content
    }
    return *this;
```

```
}
// Overloaded equality operator
int ABC::operator==(const ABC &s) const {
    // Uses string comparison function
    return strcmp(str, s.str);
}
// Overloaded + operator for string concatenation
ABC operator+(const ABC &s1, const ABC &s2) {
    ABC s3;
    s3.len = s1.len + s2.len;
    s3.max_length = s3.len;
    char *newstr = new char[s3.len + 1];
    strcpy(newstr, s1.str);
    strcat(newstr, s2.str);
    s3.str = newstr;
    return s3;
}
// Overloading the << operator
std::ostream &operator<<(std::ostream &s, const ABC &x) {
    s << "The String is: " << x.str;
    return s;
}
int main() {
    // Your code utilizing the overloaded operators can be written here
    return 0;
}
```

7.9. Assignment And Initialization

```
 class Student {
    char *name;
    int rollno;
public:
    Student() { name = new char[20]; }
    ~Student() { delete[] name; }
```

```
    Student &operator=(const Student &e);
};
```

Now, the problem is that after the execution of f(), destructors for S1 and S2 will be executed. Since both S1 and S2 point to the same storage, execution of the destructor twice will lead to an error as the storage being pointed to by S1 and S2 was disposed of during the execution of the destructor for S1 itself.

7.10. Type Conversions

We have overloaded several kinds of operators but we haven't considered the assignment operator (=). It is a very special operator with complex properties. The = operator assigns values from one variable to another or assigns the value of a user-defined object to another of the same type. For example

```
 int x, y;
x = 100;
y = x;
```

Here, first 100 is assigned to x and then x to y.

Consider another statement: 13 = t1 + t2;

This statement assigns the result of addition, which is of type Time, to object t3, also of type Time. So the assignments between basic types or user-defined types are taken care of by the compiler provided the data type on both sides of = are of the same type.

But what to do in case the variables are of different types on both sides of the = operator? In this case, we need to provide a solution to the compiler.

Three types of situations might arise for data conversion between different types:

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

Now let us discuss the above three cases:

(i) Basic Type to Class Type

This type of conversion is straightforward. For example, the following code segment converts an int type to a class type.

```
class Distance {
  int feet;
  int inches;
```

```
public:
    Distance(int dist) { //constructor
        feet = dist / 12;
        inches = dist % 12;
    }
};
```

(ii) Class Type to Basic Type

For conversion from a basic type to class type, the constructors can be used. But for conversion from a class type to basic type, constructors do not help at all. In C++, we have to define an overloaded casting operator that helps in converting a class type to a basic type.

```
class Matrix {
  // members and methods
public:
  operator float() {       float sum = 0.0;

    for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++)
        sum += a[i][j] * a[i][j];
    }
    return sqrt(sum); // norm of the matrix
  }};
```

7.11 Summary

In this lesson, we explored the concept of operator overloading and the operators that can be overloaded in C++. Despite overloading operators, their precedence remains unchanged. Additionally, we discussed the usage of '++' and '--' operators as both postfix and prefix operators, highlighting the need for separate functions to overload them for different applications.

Moreover, we emphasized that private data members of a class can only be accessed within member functions of that class.

7.12. Self Assessment Questions

1. What do you mean by dynamic binding? How it is useful in OOP?
2. What is the use of preprocessor directive #include<iostream>?
3. How does a main () function in c++ differ from main () in c?
4. Describe the major parts of a c++ program.
5. Write a program to read two numbers from the keyboard and display the larger value on the screen.
6. Write a program to input an integer value from keyboard and display on screen "WELL DONE" that many times.
7. What is the use of a constructor function in a class? Give a suitable example of a
8. Constructor function in a class.
9. Design a class having the constructor and destructor functions that shiukd display
10. The number of object being created or destroyed of this class type.

Unit 8

Inheritance

Objective:

- Introduction of OOPs,
- Single Inheritance, Multiple Inheritance,
- Multilevel Inheritance, Hierarchical inheritance,
- Hybrid Inheritance,
- Container Classes, Virtual Base Classes,
- Construction in Derived classes,
- Virtual Function, Pure Virtual Functions, Abstract Classes

Structure:

8.1. Introduction OOPs – Inheritance
8.2. Single Inheritance,
8.3. Multiple Inheritance,
8.4. Hierarchical inheritance ,
8.5. Multilevel Inheritance,
8.6. Hybrid Inheritance,
8.7. Constructor call in Multiple Inheritance in C++,
8.8. Upcasting in C++,
8.9. Functions that are never Inherited,
8.10. Inheritance and Static Functions in C++,
8.11. Hybrid Inheritance and Virtual Class in C++,
8.12. Hybrid Inheritance and Constructor call
8.13. Summary
8.14. Self Assessment Questions

8.1. Introduction OOPs – Inheritance

Inheritance allows for the reuse of code by inheriting previously written functionality. The class from which features are inherited is known as the Base class, while the class that inherits is referred to as the Derived class, or sometimes as the Parent and Child classes, respectively. When a derived class inherits from a base class, it can utilize all the functions defined in the base class, thus promoting code reuse.

For example, consider the `HumanBeing` class, which might include properties such as hands, legs, and eyes, as well as functions like walk, talk, eat, and see. Classes like `Male` and `Female` could also be defined, inheriting from the `HumanBeing` class. This inheritance allows them to inherit all the properties and functions of the `HumanBeing` class, making it easier to maintain and extend the code.

The `protected` access specifier is the last level of accessibility. It restricts the accessibility of class members to within the class itself, but these members can be accessed by any subclass, providing controlled access while preserving the encapsulation of the class.

```
class ProtectedAccess
  {
// protected access modifier protected:
int x;              // Data Member Declaration
void display(); // Member Function decaration
}
```

Inheritance allows a class to inherit properties and behaviors from another class. The class providing the properties is known as the Parent, Base, or Super class, while the class that inherits these properties is referred to as the Child, Derived, or Sub class.

This concept enables code reuse, as the derived class can use the methods and fields of the base class. By inheriting, a new class can extend or customize the behavior of the existing class without having to rewrite the code, which supports efficient and organized code management.

Inheritance also facilitates hierarchical classification. For example, a 'bird' class could serve as a base class, from which a 'flying bird' class might inherit, and further, a 'robin' class could inherit from 'flying bird'. This structure allows derived classes to share common features with their base classes, enhancing the modularity and reusability of the code.

In object-oriented programming, inheritance is a key concept that enhances reusability. It enables us to add new features to an existing class without altering its original structure. This is achieved by creating a new class that derives from the existing one, thereby inheriting all its properties and methods. The power of inheritance lies in its ability to allow programmers to reuse a class as it is, or to modify it to meet specific needs without introducing undesirable side effects into other classes. We have already explored the process of inheritance in the context of object-oriented programming, particularly in C++ .

Basic Syntax of Inheritance

"class Subclass_name : access_mode Superclass_name"

While defining a subclass like this, the superclass must be already defined or at least declared before the subclass declaration.

Access Mode is used to specify the mode in which the properties of superclass will be inherited into subclass, public, private or protected .

Example for Inheritance in C++

```
class Animal
{
public:
int legs = 4;
};
// Dog class inheriting Animal class class Dog : public Animal
{
 public:
 int tail = 1;
};
int main()
{
Dog d;
cout << d.legs; cout << d.tail;
}
```

Inheritance allows a class to include the members of other classes without repetition of members. There were three ways inheritance means, "public parts of super class remain public and protected parts of super class remain protected." Private Inheritance means "Public and Protected Parts of Super Class remain Private in Sub- Class".

Protected Inheritance means “Public and Protected Parts of Superclass remain protected in Subclass”.

A pointer is “a variable which holds a memory address”. Any variable declared in a program has two components:

(i) “Address of the variable”

(ii) “Value stored in the variable”. For example, int x = 986;

The above declaration tells the C++ compiler for :

(a) Reservation of space in memory for storing the value.

(b) Associating the name x with his memory location.

(c) Storing the value 386 at this location”. It can be represented with the following figure:

“location name” x

“value at location” 386

“location number” 3313

Here, the address 3313 is assumed one, it may be some other address also.

The pointers are one of the most useful and strongest features of C++. There are three useful reason for proper utilization of pointer :

(i) The memory location can be directly accessed and manipulated.

(ii) Dynamic memory allocation is possible.

(iii) Efficiency of some particular routines can be improved

8.1.1. Access Modifiers and Inheritance:

“Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public”.

1) Public Inheritance

“This is the most used inheritance mode. In this the protected member of super class becomes protected members of subclass and public becomes public”.

“class Subclass : public Superclass”

2) Private Inheritance

“In private mode, the protected and public members of super class become private members of derived class”.

“class Subclass : Superclass // By default its private inheritance”

3) Protected Inheritance

“In protected mode, the public and protected members of Super class become protected members of Subclass”.

“class subclass : protected Superclass”

	Derived Class	Derived Class	Derived Class
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Table showing all the Visibility Modes

In C++, inheritance represents the highest level of relationship between classes. It provides a direct and powerful means of depicting hierarchical relationships. The key strength of inheritance lies in its ability to facilitate the reuse of a class that meets most, but not all, of our requirements, and to customize it in a manner that avoids introducing unintended side effects into other classes.

To illustrate this concept effectively, it is essential to carefully review the attributes and operations of the classes involved and establish a clear inheritance relationship. This involves identifying a base class (or superclass) that encapsulates common attributes and behaviors, and then defining derived classes (or subclasses) that inherit these characteristics while potentially adding or modifying their own specific attributes and behaviors. This structured approach ensures efficient code reuse and maintenance within object-oriented programming paradigms like C++.

Inheritance relationship table

Class	Depends on
A	.....
B	A
C	A
D	B
B1	B
B2	B

“Containment relationship means the use of an object of a class as a member of another class.

This is an alternative and complimentary technique to use the class inheritance. But, it is often a tricky issue to choose between the two techniques. Normally, if there is need to override attributes or functions, then the inheritance is the best choice. On the other hand, if we want to represent a property by a variety of types, then the containment relationship is the right method to follow. Another place where we need to use an object as a member is when we need to pass an attribute of a class as an argument to the constructor of another class. The "another" class must have a member object that represents the argument. The inheritance represents is a relationship and the containment represents has a relationship".

Use relationship gives information such as the various classes a class uses and the way it uses them. For example, a class A can use classes B and C in several ways:

1. "A reads member of B"
2. "A calls a member of C"
3. "A creates B using new operator"

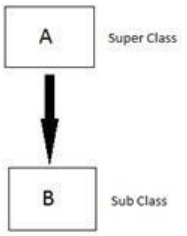
8.1.1. Types of Inheritance in C++

In C++, we have 5 different types of Inheritance.

1. "Single Inheritance"
2. "Multiple Inheritance"
3. "Hierarchical Inheritance"
4. "Multilevel Inheritance"
5. "Hybrid Inheritance (also known as Virtual Inheritance)"

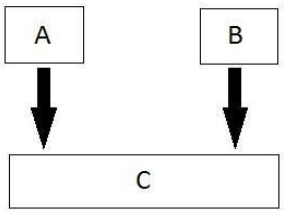
8.2. Single Inheritance in C++

In this type of inheritance "one derived class inherits from only one base class". It is the most simplest form of Inheritance.

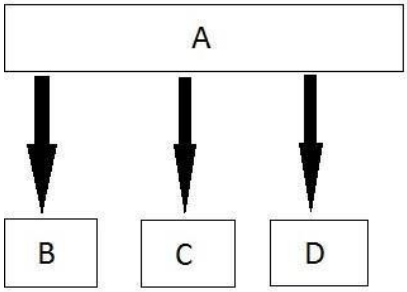


8.3. Multiple Inheritances in C++

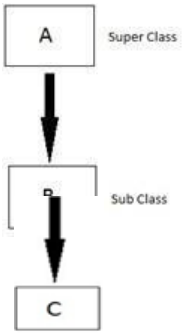
In this type of inheritance "a single derived class may inherit from two or more than two base classes".



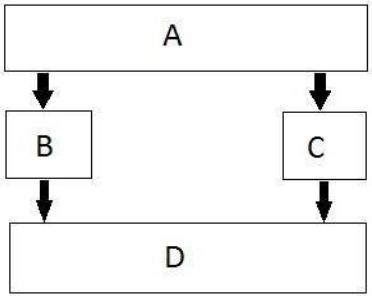
8.4. Hierarchical Inheritance in C++



8.5. Multilevel Inheritance in C++



8.6. Hybrid (Virtual) Inheritance in C++



8.6.1. Order of Constructor Call with Inheritance in C++

Base class constructors are always called in the derived class constructors. Whenever you create a derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution .

Points to Remember

1. Whether derived class's default constructor is called or parameterized is called, base class's default constructor is always called inside them .
2. To call base class's parameterized constructor inside derived class's parameterised constructor, we must mention it explicitly while declaring derived class's parameterized constructor .

Base class Default Constructor in Derived class Constructors

Default constructor is present in all the classes. In the below example we will see when and why Base class's and Derived class's constructors are called .

```
class Base
“{
   int x; public:
// default
constructor
Base()
{
 cout << "Base default constructor\n";
 }
};
class Derived : public Base
{
int y;
public:
// default
constructor
Derived()
{
cout << "Derived default constructor\n";
```

```
}
// parameterized constructor
Derived(int i)
{
cout << "Derived parameterized constructor\n";
}
};
int main()
{
Base b; Derived d1;
Derived d2(10);
}
```

8.6.2. Base class Parameterized Constructor in Derived class Constructor

“We can explicitly mention to call the Base class's parameterized constructor when Derived class's parameterized constructor is called”.

```
class Base
“{
 int x; public:
 // parameterized constructor
 Base(int i)
 {
 x = i;
 cout << "Base Parameterized Constructor\n";
 }
};
class Derived : public Base
{
 int y; public:
 // parameterized constructor
 Derived(int j):Base(j)
{
 y = j;
 cout << "Derived Parameterized Constructor\n";
```

```
 }
};
int main()
{
 Derived d(10) ;
}”
```

8.7. Constructor call in Multiple Inheritance in C++

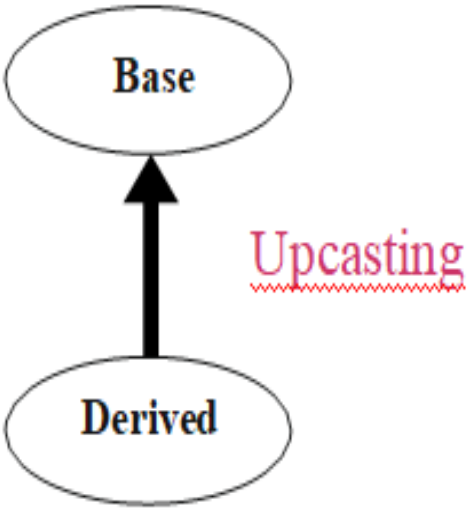
It is almost the same, “all the Base class's constructors are called inside derived class's constructor, in the same order in which they are inherited”.

```
 class A : public B, public C ;
```

In this case, “first class B constructor will be executed, then class C constructor and then class A constructor”.

8.8. Upcasting in C++

Upcasting is using the Super class's reference or pointer to refer to a Sub class's object. Or we can say that, the act of converting a Sub class's reference or pointer into its Super class's reference or pointer is called Upcasting .



```
class Super
{
int x;
public:
void funBase()
{
 cout << "Super function";
}
};
class Sub:public Super
{
 int y;
};

int main()
{
   Super* ptr;    // Super class pointer
   Sub obj;
   ptr = &obj;
   Super &ref;    // Super class's reference
   ref=obj;
}
```

The opposite of Upcasting is Downcasting, in which we convert Super class's reference or pointer into derived class's reference or pointer.

8.9 Functions that are never Inherited

- ·”Constructors and Destructors are never inherited and hence never overrided.
- ·Also, assignment operator = is never inherited. It can be overloaded but can't be inherited by sub class”.

8.10 Inheritance and Static Functions in C++

1. "They are inherited into the derived class".
2. "If you redefine a static member function in derived class, all the other overloaded functions in base class are hidden".
3. "Static Member functions can never be virtual. We will study about Virtual in coming topics".

8.11. Hybrid Inheritance and Virtual Class in C++

In Multiple Inheritance, the derived class inherits from more than one base class. Hence, in Multiple Inheritance there are a lot chances of ambiguity.

```
class A
{
void show ();
};
class B:public A
{
//class definition
};
class C:public A
{
 //class defintion
};
class D:public B, public C

  //class definition
  };
  int main()
  {
  D obj;
  obj.show();
  }
```

In this case both class B and C inherits function show() from class A. Hence class D has two inherited copies of function show(). In main() function when we call function show(), then ambiguity arises, because compiler doesn't know which show() function to call. Hence we use Virtual keyword while inheriting class .

```
class B : virtual public A
{
  //class definition
};

class C : virtual public A
{
  //class definition
};

class D : public B, public C
{
  //class definition
};
```

8.12 Hybrid Inheritance and Constructor call

In the context of hybrid inheritance, as is typically the case in C++, when an object of a derived class is instantiated, the constructors of its base classes are also called. For instance, when creating an instance of class D, the constructors of its superclasses—class B, class C, and class A—will be invoked in sequence.

However, this process can lead to multiple calls to the constructor of class A, which is often undesirable. This redundancy occurs because both class B and class C may call their own base class constructors, potentially resulting in multiple invocations of class A's constructor. To address this, C++ uses the concept of a virtual base class. When a class is virtually inherited, only a single instance of the virtual base class is created, which is shared by all derived classes. Consequently, the constructor for the base class A is called only once by the constructor of the concrete class D.

If there are multiple calls intended to initialize the base class A from within class B or class C when an object of class D is created, these redundant calls are effectively skipped, ensuring that the base class constructor is invoked only once. This mechanism prevents the issues that arise from redundant initialization and ensures that the inheritance structure remains efficient and coherent.

8.13 Summary

In this chapter, we explored the fundamentals of Object-Oriented Programming (OOP). We covered various inheritance types, including Single Inheritance, Multiple Inheritance, Multilevel Inheritance, Hierarchical Inheritance, and Hybrid Inheritance. Additionally, we discussed Container Classes, Virtual Base Classes, the construction of objects in derived classes, as well as Virtual Functions, Pure Virtual Functions, and Abstract Classes.

8.14 Self Assessment Questions

1. What is inheritance in object-oriented programming?
2. Explain the difference between single inheritance and multiple inheritance.
3. What are the advantages of using inheritance?
4. What is multilevel inheritance? Provide an example.
5. Describe hierarchical inheritance.
6. What challenges can arise with multiple inheritance, and how can they be addressed?
7. What is hybrid inheritance, and why is it used?
8. Explain the concept of virtual inheritance and its benefits.
9. How does inheritance support code reuse?
10. What is the role of the 'super' keyword in inheritance?
11. How do constructors and destructors work in derived classes?
12. What is the difference between a base class and a derived class?
13. Can a class inherit from multiple base classes in C++? How does this work?
14. What is the impact of inheritance on the accessibility of base class members in derived classes?
15. How can you prevent a class from being instantiated in C++?

Unit 9

The C++ I/O System Basics

Objective:

- Introduction C++ Streams,
- C++ Stream Classes,
- Unformatted I/O Operations,
- Formatted I/O Operations,
- Manipulators

Structure:

Introduction:

The object-oriented I/O system specified by C++ (henceforth referred to as the C++ I/O system) and the one inherited from C are the two full I/O systems that C++ provides. The I/O system in C++ is fully integrated, much like in C-based I/O. In reality, the various facets of C++'s input/output system, such as disk and console I/O, are only distinct viewpoints on the same process.

Every program follows the input-process-output cycle, taking some input data and producing processed data as output. The vast array of I/O functions available in C that can be utilized in C++ programs is supported by C++. However, these are not allowed to be used for two reasons: first, C++'s I/O methods support the OOP idea, and second, C's I/O methods are unable to accommodate user-defined data types like class objects. C++ implements I/O operations with the console and disk files using the notion of streams and stream classes.

9.1 C++ streams

A logical device that generates or consumes information is called a stream. The I/O system connects a stream to a physical device. Even while the actual physical devices that each stream is connected to may vary greatly, they all perform in the same manner. Almost any kind of physical device can use the same I/O operations since all streams behave in the same way. For instance, writing to the printer or the screen can be accomplished with the same function that writes to a file. This method has the benefit of simply requiring you to learn one I/O system.

A stream can function as a source or a destination; it is known as the input stream when data is sent to the program and as the output stream when output is received from the program.

Predefined streams called cin and cout are included in C++ and open automatically when a program runs.The input stream connected to the standard input device is represented by cin, while the output stream connected to the standard output device is represented by cout.

9.2 C++ Stream Classes

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes. Figure 9.1 shows the hierarchy of the stream classes used for input and output operations with the console unit. These classes are declared in the header file iostream. The file should be included in all programs that communicate with the console unit.

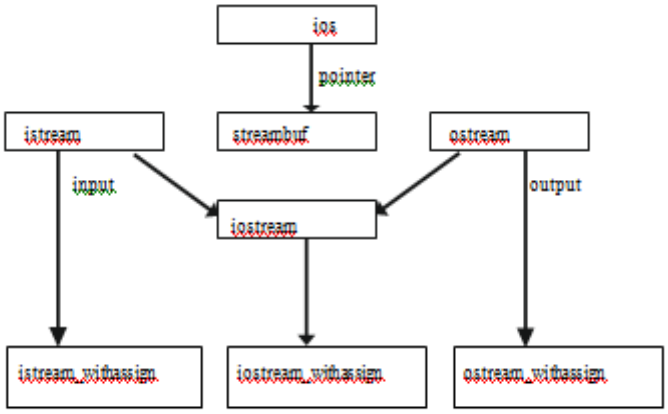


Figure 9.1 Stream classes for console I/O operations

The base class for istream (input stream) and ostream (output stream), which are base classes for iostream (input/output stream), is ios, as shown in figure 9.1.In order to prevent the iostream from inheriting more than one copy of its members, the class ios is designated as the virtual base class.

The fundamental functionality for both structured and unformatted input/output operations is provided by the class ios. Formatted and unformatted input capabilities are provided by the class istream, while formatted output facilities are provided by the class ostream (via inheritance).

The tools for managing both input and output streams are provided by the class iostream. Assignment operations are added to these classes by three classes: istream_with assign, ostream_with assign, and iostream_with assign

Table 9.1 Stream classes for console operations

Class name	Contents
ios(General input/output stream class)	Contains basic facilities that are used by all other input and output classes Also contains a pointer to buffer object(streambuf object) Declares constants and functions that are necessary for handling formatted input and output operations
istream(input stream)	Inherits the properties of ios Declares input functions such as get(),getline() and read() Contains overloaded extraction operator>>
ostream(output stream)	Inherits the property of ios Declares output functions put() and write() Contains overloaded insertion operator <<
iostream (input/output stream)	Inherits the properties of ios stream and ostream through multiple inheritance and thus contains all the input and output functions
streanbuf	Provides an interface to physical devices through buffer Acts as a base for filebuf class used ios files

9.3 Unformatted input/output Operations:

9.4.1 Overloaded operators >> and<<

Objects cin and cout are used for input and output of data by using the overloading of >> and << operators.The >> operator is overloaded in the istream class and << is overloaded in the ostream class. The following is the format for reading data from keyboard:

cin>>variable1>>variable2>>…..... >>variable n

where variable 1 to variable n are valid C++ variable names that have declared already.This statement will cause the computer to stop the execution and look for the input data from the keyboard.the input data for this statement would be
data1 data2…..... data nThe input data are separated by white spaces and should match the type of variable in the cin list spaces, newlines and tabs will be skipped.

The operator >> reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type. For example consider the code

int code; cin>> code;

Suppose the following data is entered as input 42580 the operator will read the characters upto 8 and the value 4258 is assigned to code. The character D remains in the input streams and will be input to the next cin statement. The general form of displaying data on the screen is

cout <<item1<<item2<<…..... <<item n

The item item1 through item n may be variables or constants of any basic type.

9.4.2 put() and get() functions:

The classes istream and ostream define two member functions get(),put() respectively to handle the single character input/output operations. There are two types of get() functions. Both get(char *) and get(void) prototype can be used to fetch a character including the blank space,tab and newline character. The get(char *) version assigns the input character to its argument and the get(void) version returns the input character.

Since these functions are members of input/output Stream classes, these must be invoked using appropriate objects.

Example Char c;

cin.get(c) //get a character from the keyboard and assigns it to c while(c!='\n')

{ cout<< c; //display the character on screen cin.get(c) //get another character

}

This code displays a line of text after reading it. While reading a character, the operator >> will omit newline characters and white spaces. The while loop above won't function correctly if the expression

The statement **cin >> c;** is used in place of **cin.get(c);**. The **get()** version is used as follows:

char c;

c = cin.get();

This statement reads a single character from the standard input and assigns it to the variable **c.**

The value returned by the function **get()** is assigned to the variable **c**. The function **put()**, a member of the **ostream** class, can be used to output a line of text character by character. For

example:

cout.put('x');

displays the character 'x', and

cout.put(ch);

displays the value of the variable ch.

The variable **ch** must contain a character value. A number can also be used as an argument to the **put()** function. For example:

cout.put(68);

displays the character 'D'. This statement will convert the numeric value 68 to a character value and display the character whose ASCII value is 68.

The following segment of a program reads a line of text from the keyboard and displays it on the screen:

```
char c;
        cin.get(c);
        while (c != '\n') {
           cout.put(c);
           cin.get(c);
        }
```

This code reads characters from the standard input one by one until a newline character is encountered, and then it outputs each character to the standard output.

The program 9.1 illustrates the use of two character handling functions.

Program 9.1: Character I/O with get() and put()"

```
#include      <iostream>
using namespace std; int
main()
{
int count=0; char c;
cout<<"INPUT TEXT \n"; cin.get( c
);
while ( c 1='\n' )
{ cout.put( c); count++; cin.get( c );
}
```

```
cout<< “\n Number of characters =” <<count <<”\n”; return 0;
}
```

Input

Object oriented programming Output

Object oriented programming

Number of characters=27

9.4.3 write() and getline() functions:

A line of text can be read or displayed effectively using the line oriented input/output functions getline() and write() .The getline() function reads a whole line of text that ends with a newline character. This function can be invoked by using the object cin as follows:

```
cin.getline(line,size);
```

This function call invokes the function getline() which reads character input into the variable line.The reading is terminated as soon as either the newline character '\n' is encountered or size-1 characters are read(whichever occurs first) .The newline character is read but not saved.instead it is replaced by the null character.For example consider the following code
char name[20]; cin.getline(name,20);
Assume that we have given the following input through keyboard: Bjarne Stroustrup<press Return>
The character array name will be allocated to this input after it has been appropriately read.
Assume the following input for the moment:
Press Return to exit object-oriented programming.
In this instance, the input will end after viewing the subsequent Nineteen characters Pro Object-Oriented

Recall that the two blank spaces in the string are also considered.Operator >> can be used to read strings in the following ways:
cin>>name;
However, keep in mind that Cin can read strings devoid of white spaces.This indicates that Cin is only able to read a single word, not a string of words like "Bjarne Stroustrup."However, it can accurately read the following string:
Bjarne_Stroustrup

After reading the string ,cin automatically adds the terminating null character to the character array.

The program 9.2 demonstrates the use of >> and getline() for reading the strings.

Program 9.2: Reading Strings With getline() #include
<iostream>
using namespace std; int main()
{ int size=20; char city[20]; cout<<"enter city name:\n "; cin>>city;
cout<<"city name:"<<city<<"\n\n"; cout<<"enter city name again: \n"; cin.getline(city,size);
cout<<"city name now:"<<city<<"\n\n";
cout<<"enter another city name: \n";
cin.getline(city,size);
cout <<"New city name:"<<city<<"\n\n'; return 0;
}
output would be:

first run
enter city name:
Delhi
Enter city name again:
City name now:
Enter another city name:
Chennai
New city name: Chenna
When the getline() function is called for the first time, it reads the newline character "\n\" that ends "Delhi" and answers the request to "enter city name again" without waiting for a response. The symbol "\n" signifies a line that is blank.

The write() method takes the following form and shows a single line:
"cout.write(line,size)"

The name of the string to be displayed is represented by the first parameter line, and the number of characters that are automatically displayed when the null character is encountered

is indicated by the second argument size. It displays outside of the line boundary if the size exceeds the length of the line.Program 9.3 shows how to display a string using the write() method.

Program 9.3

Displaying String With write()

```
#include <iostream>
#include<string> using
namespace std; int main(0
{

char * string1=”C++”;
char * string2 =”Programming”; int
m=strlen(string1);
int n =strlen(string2); for (int
i=1;i<n;i++)
{
cout.write(string2,i);
cout<<”\n”;
}
for (i<n;i>0;i--)
{
cout.write(string2,i);
cout<<”\n”;}
//concatenating strings
cout.write(string1,m).write(string2,n); cout<<”\n”;
//crossing the boundary
cout.write(string1,10); return 0;
}
output:
```

```
P
Pr
Pro
Prog
Progr
Progra
Program
Programm
Programmi
Programmin
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr
P
```

The last line of the output indicates that the statement

cout.write(string1,10);

displays more character than what is contained in string1.

It is possible to concatenate two strings using the write() function. The statement

cout.write(string1,m).write(string2,n);

is equivalent to the following two statements:

cout.write(string1,m);

cout.write(string2,n);

9.5 Formatted Console I/O Operations:

C++ provides several features for formatting output, including:

- ios class functions and flags.
- Manipulators.
- User-defined output functions.

The ios class contains numerous member functions that facilitate formatting output in various ways. Some of the most important ones are listed in Table 9.2.

Table 9.2 ios format functions

Function	Task
Width()	To specify the required field size for displaying an output value
Precision()	To specify the number of digits to be displayed after the decimal point of float value
Fill()	To specify a character that is used to fill the unused portion of a field
Setf()	To specify format flags that can control the form of output display(such as left-justification and right-justification)
Unsetf()	To clear the flags specified

Special functions called manipulators can be added to I/O statements in order to change the stream's format parameter. A few key and often used manipulator functions are displayed in Table 9.3. The program needs to include the file iomanip in order to access these manipulators

Table 9.3 Manipulators

Manipulators	Equivalent ios function
setw()	width()
setprecision()	precision()
setfill()	fill()
setiosflags()	setf()
resetiosflags()	unsetf()

We can write our own manipulator functions in addition to these common library manipulators to offer any unique output formats.

9.6 Managing Output with Manipulators

Defining Field Width:width():

The width() function is used to define the width of a field necessary for the output of an item.As it is a member function object is required to invoke it like

cout.width(w);

here w is the field width.The output will be printed in a field of w character wide at the right end of field.The width() function can specify the field width for only one item(the item that follows immediately).After printing one item(as per the specification) it will revert back the default.for example,the statements

cout.width(5); cout<<543<<12<<"\n";

will produce the following output:

		5	4	3	1	2

The value 543 is printed right justified in the first five columns.The specification width (5) does not retain the setting for printing the number 12.this can be improved as follows:

```
cout.width(5);
cout<<543;
cout.width(5);
```

cout<<12<<"\n";

This produces the following output:

		5	4	3				1	2

The field width should be specified for each item.C++ never truncate the values and therefore, if the specified field width is smaller than the size of the value to be printed,C++ expands the field to fit the value.program 9.4 demonstrates how the function width() works

Program 9.4: Specifying field size with width()

```
#include <iostream> using
namespace std; int main()
{
int item[4] ={ 10,8,12,15};
int cost[4]={75,100,60,99};
cout.width(5); cout<<"Items";
cout.width(8); cout<<"Cost";
cout.width(15); cout<<"Total
Value"<<"\n"; int sum=0;
for(int i=0;i<4 ;i++)
{
cout.width(5); cout<<items[i];
cout.width(8); cout<<cost[i];
int value = items[i] * cost[i];
cout.width(15); cout<<value<<"\n";
sum= sum + value;
}
cout<<"\n Grand total = ";
cout.width(2); cout<<sum<<"\n";
return 0;
}
```

The output of program 9.4 would be

ITEMS	COST	TOTAL VALUE
10	75	750
8	100	800
12	60	720
15	99	1485

Grand total =3755

9.6.1 Setting Precision: precision()

By default, the floating numbers are printed with six digits after the decimal points. However, we can specify the number of digits to be displayed after the decimal point while printing the floating point numbers.

This can be done by using the precision () member function as follows:

cout.precision(d);

where d is the number of digits to the right of decimal point.for example the statements

cout.precision(3); cout<<sqrt(2)<<"\n";

cout<<3.14159<<"\n";

cout<<2.50032<<"\n";

will produce the following output:

1.141 (truncated) 3.142(rounded to nearest cent) 2.5(no trailing zeros)

Unlike the function width(),precision() retains the setting in effect until it is reset. That is why we have declared only one statement for precision setting which is used by all the three outputs. We can set different valus to different precision as follows:

```
cout.precision(3);

cout<<sqrt(2)<<"\n";

cout.precision(5);

cout<<3.14159<<"\n";
```

We can also combine the field specification with the precision setting.example:

```
cout.precision(2);
cout.width(5);

cout<<1.2345;
```

	1		2	3

Program 9.5: PRECISION SETTING WITH precision()

```
 #include<iostream>
#include<cmath> using
namespace std; int main()
{
cout<<"precision set to 3 digits\n\n";
cout.precision(3);
cout.width(10); cout<<"value";
cout.width(15); cout<<"sqrt_of
_value"<<"\n"; for (int n=1;n<=5;n++)
{
cout.width(8); cout<<n;
cout.width(13);
cout<<sqrt(n)<<"\n";
}
cout<<"\n precision set to 5 digits\n\n";
cout.precision(5);
cout<<"sqrt(10) = " <<sqrt(10)<<"\n\n";
cout.precision(0);
cout<<"sqrt(10) = " <<sqrt(10)<<"(default setting)\n"; return
0;
}
```

The output is Precision set
to 3 digits

The output is Precision set to 3 digits

VALUE	SQRT OF VALUE
1	1
2	1.41
3	1.73
4	2
5	2.24

Precision set to 5 digits

Sqrt(10)=3.1623

Sqrt(10)=3.162278 (Default setting)

9.6.2 FILLING AND PADDING :fill()

The unused portion of field width is filled with white spaces, by default. The fill() function can be used to fill the unused positions by any desired character.It is used in the following form:

cout.fill(ch);

Where ch represents the character which is used for filling the unused positions.Example:

cout.fill('*'); cout.width(10);

cout<<5250<<"\n"; The output

would be:

*	*	*	*	*	*	5	2	5	0

FORMATTING FLAGS, Bit Fields and setf():

The setf() , a member function of the ios class, can provide answers left justified. The setf() function can be used as follows:

cout.setf(arg1.arg2)

The arg1 is one of the formatting flags defined in the class ios. The formatting flag specifies the format action required for the output. Another ios constant,arg2,known as bit field specifies the group to which the formatting flag belongs. for example:

cout.setf(ios::left,ios::adjustfield); cout.setf (ios::scientific,ios::floatfield);

Note that the first argument should be one of the group members of the second argument.

Consider the following segment of code:

```
cout.fill('*');

cout.setf(ios::left,ios::adjustfield);

cout.width(15); cout<<"table1"<<"\n";
```

This will produce the following output:

						1	*	*	*	*	*	*	*	*

The statements cout.fill('*');

```
cout.precision(3);
cout.setf(ios::internal,ios::adjustfield);
cout.setf(ios::scientific,ios::floatfield);
cout.width(15);
cout<<-12.34567<<"\n";
```

Will produce the following output

-	*	*	*	*	*	1	.	2	3	5	e	+	0	1

"The iomanip header file in C++ offers a range of functions known as manipulators, which are utilized to manipulate the format of output. These manipulators provide similar functionalities as those of the ios member functions and flags.

For instance, multiple manipulators can be chained together in a single statement, allowing for concise formatting of output. This chaining is particularly useful when displaying

multiple columns of output:

The most commonly used manipulators are shown below:

Manipulator	Meaning	Equivalent
setw(int w)	Set the field width to w	width()
setprecision(int d)	Set the floating point precision to d	precision()
setfill(int c)	Set the fill character to c	fill()
setiosflags(long f)	Set the format flag f	setf()
resetiosflags(long f)	Clear the flag specified by f	unsetf()
endl	Insert new line and flush stream	"\n"

Examples of manipulators are given below:

```
cout << setw(10) << 12345;
```

This statement prints the value **12345** right-justified in a field of 10 characters. To left-justify the output, you can modify the statement as follows:

```
cout << setw(10) << setiosflags(ios::left) << 12345;
```

One statement can be used to format output for multiple values. For example, the statement:

```
cout << setw(5) << setprecision(2) << 1.2345
     << setw(10) << setprecision(4) << sqrt(2)
     << setw(15) << setiosflags(ios::scientific) << sqrt(3)
     << endl;
```

This will print all three values in one line with field widths of 5, 10, and 15 respectively. The following program illustrates the formatting of output values using both manipulators and **ios** functions:

```
#include <iostream>
#include <iomanip>

using namespace std;
```

```
int main() {
    cout.setf(ios::showpoint);
    cout << setw(5) << "n" << setw(15) << "inverse of n" << setw(15) << "sum of
terms" << endl;

    double term, sum = 0;

    for (int n = 1; n <= 10; n++) {
        term = 1.0 / static_cast<float>(n);
        sum += term;

        cout << setw(5) << n
            << setw(14) << setprecision(4) << setiosflags(ios::scientific) << term
            << setw(13) << resetiosflags(ios::scientific) << sum << endl;
    }

    return 0;
}
```

In this program, **setw, setprecision, setiosflags,** and **resetiosflags** are used to control the formatting of the output, demonstrating the use of manipulators and **ios** functions effective

9.7 Designing our own Manipulator Functions

The general form for creating a manipulator without any argument is ostream & manipulator (ostream & output)

```
{
        ……………
        …..... (code)
        ……..
        return output;
        }
```

The following program illustrates the creation and use of user defined manipulators.

The construction and application of user-defined manipulators are demonstrated in the following program.

```
“#include <iostream>
#include <iomanip>
using namespace std;
ostream &currency (ostream & output)
{
output<< “Rs”; return
output;
}
ostream & form (ostream &output)
{
        output.set(ios::showpos);

        output.setf(ios::showpoint); output.fill('*');
        output.precision (2);
        output<<setiosflags(ios::fixed)<<setw(10); return output;
}
int main()
{
cout<<currency<<form<<7864.5; return 0;
}
```

the output of program is

Rs**+7864.50

The program form represents a complex set of format functions and manipulators.

9.8 Summary

A stream in C++ is essentially a sequence of bytes that serves as a source or destination for input/output (I/O) data operations. There are two primary types of streams: input streams, which provide data to the program, and output streams, which receive output from the program.

In C++, the I/O system includes a hierarchy of stream classes, which are declared in the iostream header file. These classes facilitate input and output operations.

The cin stream represents the input stream connected to the standard input device, typically the keyboard, while cout represents the output stream connected to the standard output device, usually the console or terminal.

In C++, the >> operator is overloaded in the istream class to function as an extraction operator, allowing data to be extracted from the input stream. Conversely, the << operator is overloaded in the ostream class to function as an insertion operator, enabling data to be inserted into the output stream.

For more efficient handling of text input and output, C++ provides line-oriented I/O functions such as getline() for reading a line of text from the input stream, and write() for writing a line of text to the output stream.

Additionally, the iomanip header file offers a collection of manipulator functions that can be used to manipulate output formats, providing greater flexibility and control over the appearance of output data.

9.9 Self Assessment Questions

- What is a stream?
- Describe briefly the features of the I/O system supported by C++.
- How is cout able to display various types of data without any special instructions?
- Why is it necessary to include the file iostream in all our programs?
- What is the role of an iomanip file?
- What is the basic difference between manipulators and ios member functions in implementation? Give examples.

Unit: 10

Introduction to JAVA

Objective:

- Understanding of basic concepts of Java.
- Grasp fundamental programming concepts like variables, data types, operators, control flow statements (if/else, loops), and functions/methods.
- Understand how to write well-structured, readable, and maintainable Java code.
- Be able to compile and run Java programs using an Integrated Development Environment (IDE).

Structure:

10.1 History of Java

Java's journey began in 1991 with a team of Sun Microsystems engineers known as the Green Team. Led by James Gosling, Mike Sheridan, and Patrick Naughton, they envisioned a new language for consumer electronics like set-top boxes. Initially called "Greentalk" with a ".gt" file extension, it later became "Oak."

Though intended for embedded systems, the project took an unexpected turn. The rise of the internet in the early 1990s presented a new opportunity. Netscape, a leading web browser at the time, saw Java's potential and incorporated the technology. This shifted the focus towards internet programming.

Today, Java's applications extend far beyond its internet roots. It's used in mobile devices, game development, e-commerce solutions, and a variety of other areas.

10.2 Features of JAVA

Java, a popular programming language renowned for its versatility and portability, offers a range of features that contribute to its widespread adoption across various domains. Here are some key features of Java:

1. **Simple and Easy to Learn**: Java was designed with simplicity in mind, making it easy for programmers to learn and use. Its syntax is similar to C and C++, reducing the learning curve for developers familiar with these languages.
2. **Platform Independence**: One of Java's most significant features is its platform independence. Java programs can run on any device that has a Java Virtual Machine (JVM), allowing them to be executed on different platforms without modification.
3. **Object-Oriented**: Java is a fully object-oriented programming language, which means it supports principles like inheritance, encapsulation, polymorphism, and abstraction. This paradigm promotes modular and reusable code, making development more manageable and scalable.
4. **Robustness**: Java is known for its robustness, primarily due to its strong memory management, exception handling, and type safety features. The language includes automatic garbage collection, which helps in managing memory efficiently and prevents memory leaks.
5. **Security**: Security is a top priority in Java. It incorporates various security mechanisms, such as a robust security model, bytecode verification, and a sandbox environment for executing untrusted code. These features make Java applications more resistant to security threats.

6. **Portability**: Java's "write once, run anywhere" (WORA) principle enables developers to write code on one platform and execute it on any other platform with a compatible JVM. This portability makes Java suitable for developing cross-platform applications.
7. **High Performance**: While Java is not as fast as low-level languages like C or C++, it offers good performance through just-in-time (JIT) compilation and optimization techniques. Modern JVMs can dynamically compile Java bytecode into native machine code at runtime, improving execution speed.
8. **Multithreading**: Java provides built-in support for multithreading, allowing developers to create concurrent, multi-threaded applications easily. The Thread class and Runnable interface facilitate the creation and management of threads, enabling efficient utilization of system resources.
9. **Rich Standard Library**: Java comes with a comprehensive standard library (Java API) that provides a wide range of pre-built classes and packages for various tasks, such as I/O operations, networking, GUI development, database connectivity, and more. This extensive library simplifies development and accelerates time-to-market for Java applications.
10. **Community Support**: Java boasts a vast and active community of developers, enthusiasts, and organizations that contribute to its growth and evolution. The availability of resources, forums, tutorials, and open-source libraries makes it easier for developers to collaborate, learn, and resolve issues.

10.3 Java Comments

In Java, comments are special lines of text ignored by the computer when running the program. These comments act like notes left within the code to explain different parts. They can be used for:

Clarity: Adding comments to your code clarifies what variables, methods, or classes do, making the code easier to understand for yourself and others.

Documentation: Comments can serve as documentation, explaining the purpose and functionality of different code sections.

Temporary Code Hiding: While not ideal, comments can temporarily disable code by placing them over the unwanted lines. This is a practice to use with caution as it's better to have proper code removal methods.

10.3.1 Types of Java Comments

- **Single-Line Comments**
- **Multi-Line Comments**
- **Javadoc Comments**

10.3.1.1 Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

```
1. //This is single line comment
```

Example:

```
public class CommentExample1 {
public static void main(String[] args) {
int i=10;//Here, i is a variable
System.out.println(i);
}
}
```

10.3.1.2 Java Multi Line Comment

The multi-line comment is used to comment multiple lines of code.

Syntax:

```
/*This is multi l
ine comment */
```

Example:

```
public class CommentExample2 {
public static void main(String[] args) {
```

```
/* Let's declare and
print variable in java. */
inti=10;
System.out.println(i);
} }
```

Output:

10

Data Types

In Java, data types specify the type of data that a variable can hold. Java supports two categories of data types: primitive data types and reference data types. Here's an overview of each:

10.4.1 Primitive Data Types:

- Numeric Types:
  - byte: 8-bit signed integer (-128 to 127)
  - short: 16-bit signed integer (-32,768 to 32,767)
  - int: 32-bit signed integer (-2^31 to 2^31-1)
  - long: 64-bit signed integer (-2^63 to 2^63-1)
  - float: 32-bit floating point (single precision)
  - double: 64-bit floating point (double precision)
- Boolean Type:
  - boolean: Represents true or false values
- Character Type:
  - char: 16-bit Unicode character (0 to 65,535)

Reference Data Types:

- Class Types: User-defined data types created using classes.
- Interface Types: Similar to class types but defined using interfaces.
- Array Types: Arrays that hold multiple values of the same type.
- Enumeration Types: Special types used to define a set of constants.

Here's an example demonstrating the declaration and initialization of variables using different data types:

Java has 8 primitive datatypes:

10.4.2 Non-Primitive Datatypes

Non-primitive datatypes are also called reference types because they reference objects. These include:

1. **Strings**
   - Sequence of characters.
   - Example: `String str = "Hello, World!";`
2. **Arrays**
   - Collection of similar types of elements.
   - Example: `int[] arr = new int[10];`

3. **Classes**
   - Blueprint from which objects are created.
   - Example: `class MyClass { int x; }`

4. **Interfaces**
   - Abstract types used to specify the behavior that classes must implement.
   - Example: `interface MyInterface { void myMethod(); }`

5. **Enumerations (Enums)**
   - Special class representing a group of constants (unchangeable variables).
   - Example: `enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }`

10.5 Type Conversion

Java supports both automatic type conversion and explicit type casting:

10.5.1 Implicit Type Conversion (Widening Conversion):

- Automatically converts a smaller datatype to a larger datatype.
- Example: `int a = 100; long b = a; // no explicit cast needed`

10.5.2 Explicit Type Conversion (Narrowing Conversion):

- Requires explicit cast to convert a larger datatype to a smaller datatype.
- Example: `double a = 10.5; int b = (int) a; // explicit cast needed`

10.6 Variables

In Java, a variable is a container that holds data which can be changed during the execution of a program. Variables are an essential concept in Java programming and are used to store information that can be referenced and manipulated within a Java application.

10.6.1 Types of Variables

1. Local Variables

- Scope: Declared inside a method, constructor, or block.
- Lifetime: Exist only within the method, constructor, or block they are declared in.
- Default Value: No default value, must be initialized before use.

```
public void myMethod() {
    int localVar = 10; // local variable
    System.out.println(localVar);
}
```

2. Instance Variables (Non-Static Fields)

Scope: "Declared in a class but outside a method, constructor, or block."

- Lifetime: Exist as long as the object of the class exists.
- Default Value: Default values (e.g., 0 for int, null for objects) are provided.

```
public class MyClass {
    int instanceVar; // instance variable

    public void myMethod() {
        instanceVar = 20;
        System.out.println(instanceVar);
    }
}
```

3. Class Variables (Static Fields)

- Scope: “Declared with the static keyword in a class, but outside a method, constructor, or block.”
- Lifetime: Exist for the entire duration of the program and are shared among all instances of the class.
- Default Value: Default values are provided.

```
public class MyClass {
    static int classVar; // class variable
    public void myMethod() {
        classVar = 30;
        System.out.println(classVar);
    }
}
```

10.6.2 Variable Declaration

A variable declaration includes a datatype and the variable name. Optionally, a variable can be initialized at the time of declaration.

```
int myVar;          // declaration
myVar = 100;        // initialization
int anotherVar = 200; // declaration and initialization
```

10.6.3 Variable Naming Rules

1. Case-Sensitive: Variable names are case-sensitive.

2. Letters and Digits: Must begin with a letter (a-z, A-Z), the dollar sign `$`, or the underscore `_`. Subsequent characters may be digits, letters, underscores, or dollar signs.

3. No Keywords: Cannot be a Java keyword.

4. Convention: Typically, variables start with a lowercase letter and follow camelCase.

10.6.4 Variable Initialization

Variables must be initialized before they are used. Java does not initialize local variables by default, so they must be explicitly initialized.

```
int myVar;          // declaration
myVar = 10;         // initialization
System.out.println(myVar); // usage
```

10.6.5 Variable Scope

The scope of a variable determines where it can be accessed and modified.

1. Local Variables: Limited to the block of code where they are declared.

2. Instance Variables: Accessible within all methods of the class where they are declared.

3. Class Variables: Accessible within all methods of the class and can be accessed without creating an instance of the class.

10.7 Constants

Constants are variables whose values cannot be changed once initialized. They are declared using the `final` keyword.

```
final int CONSTANT_VAR = 100;
```

10.8 Operators in java

In Java, operators are unique symbols designed to execute specific operations on one, two, or three operands, yielding a result. Java operators can be categorized into various types:

10.8.1. Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations.

Addition (`+`): Adds two operands.

```
int a = 10 + 5; // 15
```

Subtraction (`-`): Subtracts the second operand from the first.

```
int a = 10 - 5; // 5
```

Multiplication (`*`): Multiplies two operands.

```
int a = 10 * 5; // 50
```

Division (`/`): Divides the first operand by the second.

```
int a = 10 / 5; // 2
```

Modulus (`%`): Returns the remainder when the first operand is divided by the second.

```
int a = 10 % 3; // 1
```

10.8.2 Unary Operators

Unary operators operate on a single operand.

Unary Plus (`+`): Indicates positive value (redundant, as numbers are positive by default).

```
int a = +5; // 5
```

Unary Minus (`-`): Negates an expression.

```
int a = -5; // -5
```

Increment (`++`): Increases the value of a variable by 1.

```
int a = 5;
a++; // 6
```

Decrement (`--`): Decreases the value of a variable by 1.

```
int a = 5;
a--; // 4
```

Logical Complement (`!`): Inverts the value of a boolean expression.

```
boolean a = true;
a = !a; // false
```

10.8.3 Assignment Operators

Assignment operators assign values to variables.

Simple Assignment (`=`): Assigns the right-hand operand to the left-hand operand.

```
int a = 5; // a is 5
```

Addition Assignment (`+=`): It adds the right-hand operand to the left-hand operand and assigns the resulting value to the left-hand operand.

```
int a = 5;
a += 3; // a is 8
```

Subtraction Assignment (`-=`): It subtracts the right-hand operand from the left-hand operand and assigns the resulting value to the left-hand operand.

```
int a = 5;
a -= 3; // a is 2
```

Multiplication Assignment (`*=`): Multiplies the left-hand operand by the right-hand operand and assigns the result to the left-hand operand.

```
int a = 5;
a *= 3; // a is 15
```

Division Assignment (`/=`): Divides the left-hand operand by the right-hand operand and assigns the result to the left-hand operand.

```
int a = 5;
a /= 3; // a is 1 (integer division)
```

Modulus Assignment (`%=`): Takes the modulus using two operands and assigns the result to the left-hand operand.

```
int a = 5;
a %= 3; // a is 2
```

10.8.4 Relational Operators

Relational operators compare two operands and return a boolean result.

Relational operators in Java are used to establish relationships between variables or operands. They evaluate the relationship between two operands and return a boolean value, true or false, based on whether the relationship holds true or not. These operators are also known as comparison operators.

Here are the relational operators in Java:

1. **Equal to (==)**:
    - Checks if the values of two operands are equal.
    - Returns true if the values are equal, otherwise returns false.

- Example: a == b

2. **Not equal to (!=)**:
    - Checks if the values of two operands are not equal.
    - Returns true if the values are not equal, otherwise returns false.
    - Example: a != b
3. **Greater than (>)**:
    - Checks if the value of the left operand is greater than the value of the right operand.
    - Returns true if the left operand is greater, otherwise returns false.
    - Example: a > b
4. **Less than (<)**:
    - Checks if the value of the left operand is less than the value of the right operand.
    - Returns true if the left operand is less, otherwise returns false.
    - Example: a < b
5. **Greater than or equal to (>=)**:
    - Checks if the value of the left operand is greater than or equal to the value of the right operand.
    - Returns true if the left operand is greater than or equal to the right operand, otherwise returns false.
    - Example: a >= b
6. **Less than or equal to (<=)**:
    - Checks if the value of the left operand is less than or equal to the value of the right operand.
    - Returns true if the left operand is less than or equal to the right operand, otherwise returns false.
    - Example: a <= b

These relational operators are commonly used in decision-making statements (e.g., if, while, for) and are essential for controlling the flow of a program based on conditions. They compare the operands on either side and determine the truth value of the expression.

10.8.5 Logical Operators

Logical operators are used to combine multiple boolean expressions or values and return a boolean result.

Logical AND (`&&`): Returns true if both operands are true.

```
boolean result = (true && false); // false
```

Logical OR* (`||`): Returns true if at least one operand is true.

```
boolean result = (true || false); // true
```

Logical NOT (`!`): Returns the opposite boolean value of the operand.

```
boolean result = !true; // false
```

10.8.6 Bitwise Operators

Bitwise operators perform operations on individual bits of integer types.

Bitwise AND (&):

- Performs a bitwise AND operation on each pair of corresponding bits.
- If both bits are 1, the result is 1. Otherwise, the result is 0.
- Example: a & b

Bitwise OR (|):

- Performs a bitwise OR operation on each pair of corresponding bits.
- If either bit is 1, the result is 1. Otherwise, the result is 0.
- Example: a | b

Bitwise XOR (^):

- Performs a bitwise XOR (exclusive OR) operation on each pair of corresponding bits.
- The result is 1 if the bits are different, and 0 if they are the same.
- Example: a ^ b

Bitwise NOT (~):

- Performs a bitwise NOT (complement) operation on each bit, flipping 0s to 1s and 1s to 0s.
- Example: ~a

Left Shift (<<):

- Shifts the bits of the left operand to the left by a specified number of positions.

- Zeros are shifted in from the right.
- Example: a << n

Right Shift (>>):

- Shifts the bits of the left operand to the right by a specified number of positions.
- For positive numbers, zeros are shifted in from the left.
- For negative numbers, ones are shifted in from the left (sign extension).
- Example: a >> n

Unsigned Right Shift (>>>):

- Shifts the bits of the left operand to the right by a specified number of positions.
- Zeros are shifted in from the left, and the leftmost bits are discarded.
- Example: a >>> n

Left Shift (`<<`): Shifts the bits of the left operand left by the number of positions specified by the right operand.

```
int result = 5 << 1; // 10 (0101 << 1 = 1010)
```

10.8.7 Ternary Operator

The ternary operator, also known as the conditional operator, is a unique operator in Java that allows you to make concise decisions based on a condition. It's often used as a shorthand for simple if-else statements, providing a compact way to express conditional logic.

```
condition ? expression1 : expression2
```

10.8.8 Instance of Operator

The instanceof operator in Java is used to test whether an object is an instance of a particular class or interface. It returns either true or false based on whether the object is an instance of the specified type.

Instance of (`instanceof`): Checks if an object is an instance of a specific class.

```
String str = "Hello";
boolean result = str instanceof String; // true
```

10.9 Control statements

Control statements in Java are used to manage the flow of a program by determining which statements to execute and when. They are essential for making decisions, repeating code

blocks, and controlling the flow of loops and switch statements. Control statements in Java can be categorized into three main types: decision-making statements, iteration statements, and jump statements.

10.9.1 Decision-Making Statements

Decision-making statements in Java are used to evaluate conditions and decide which block of code to execute. There are several types of decision-making statements:

If Statement: The if statement is used to evaluate a condition and execute a block of code if the condition is true. It can also have an else block that is executed if the condition is false

```
if (condition) {
   // code to execute if condition is true
} else {
   // code to execute if condition is false
}
```

If-Else-If Ladder: This is a chain of if-else statements where the program checks multiple conditions and executes different blocks of code based on the conditions.

Syntax:

```
if (condition1) {
   // code to execute if condition1 is true
} else if (condition2) {
   // code to execute if condition1 is false and condition2 is true
} else {
   // code to execute if all conditions are false
}
```

Nested If Statement: This is an if statement inside another if statement. It is used to evaluate multiple conditions and execute different blocks of code based on the conditions.

Syntax:

```
if (condition1) {
   if (condition2) {
      // code to execute if both conditions are true
   } else {
      // code to execute if condition1 is true but condition2 is false
```

```
    }
} else {
    // code to execute if condition1 is false
}
```

Switch Statement

The switch statement in Java is a control flow statement that allows a variable to be tested for equality against a list of values. It provides a more concise way to write multiple if-else-if statements for comparing the value of a variable with multiple possible values.

Syntax:

```
switch (expression) {
    case value1:
        // code to execute if expression equals value1
        break;
    case value2:
        // code to execute if expression equals value2
        break;
    default:
        // code to execute if expression does not match any case
}
```

10.9.1 Iteration Statements

Iteration statements in Java are used to repeat a block of code until a specified condition is met. There are three types of iteration statements:

For Loop: The for loop is used to execute a block of code repeatedly for a specified number of iterations. It can be used to iterate over arrays and collections.

Syntax:

```
for (initialization; condition; increment) {
    // code to execute
}
```

While Loop: The while loop is used to execute a block of code repeatedly while a specified condition is true. It is used to repeat a block of code until a condition is met.

Syntax:

```
while (condition) {
   // code to execute
}
```

Do-While Loop: The do-while loop is similar to the while loop but the block of code is executed at least once before the condition is checked.

Syntax:

```
do {
   // code to execute
} while (condition);
```

10.9.3 Jump Statements

Jump statements in Java are used to transfer the control of the program to another part of the code. There are two types of jump statements:

Break Statement: The break statement is used to terminate a loop and transfer the control to the next statement outside the loop.

Syntax:

```
for (int i = 0; i < 10; i++) {
   if (i == 5) {
      break;
   }
   System.out.println(i);
}
```

Syntax:

```
for (int i = 0; i < 10; i++) {
   if (i % 2 == 0) {
      continue;
   }
   System.out.println(i);
}
```

10.10 Class & Object

10.10.1 Class

A class in Java is a template or blueprint that defines the characteristics and behavior of an object. It is a logical entity that does not occupy memory until an object is created from it. A class can contain:

Data Members (Instance Variables): These are variables that are declared inside the class but outside any method. They are also known as instance variables because each object of the class has its own copy of these variables.

Methods: These are blocks of code that perform specific actions. Methods can be used to expose the behavior of an object.

Constructors: These are special methods that are used to initialize objects when they are created.

Nested Classes: These are classes that are defined inside another class.

Interfaces: These are abstract classes that define a set of methods that must be implemented by any class that implements the interface.

10.10.2 Object

An object in Java is an instance of a class. It is a physical entity that has its own set of attributes (data members) and methods. Each object of a class has its own set of attributes and methods, which are separate from those of other objects of the same class.

Key Differences

Memory Allocation: A class does not occupy memory until an object is created from it. An object, on the other hand, occupies memory as soon as it is created.

Physical vs. Logical: A class is a logical entity, while an object can be both physical (e.g., a car) and logical (e.g., a banking system).

Creation: A class can only be declared once, but objects can be created multiple times from the same class.

State and Behavior: An object has both state (attributes) and behavior (methods), while a class only defines the blueprint for these.

Example

```
// Define a class
class Student {
   int id;
```

```
    String name;
    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public void displayInfo() {
        System.out.println("ID: " + id);
        System.out.println("Name: " + name);
    }
}

// Create objects
Student s1 = new Student(101, "Ravi");
Student s2 = new Student(102, "Sonoo");

// Use objects
s1.displayInfo();
s2.displayInfo();
```

In this example, Student is a class that defines the attributes id and name and the method displayInfo(). Two objects, s1 and s2, are created from the Student class, each with its own set of attributes. The displayInfo() method is used to print the attributes of each object

10.11 Array

In Java, an array is a data structure that stores a fixed-size collection of elements of the same data type. Here is an explanation based on the provided sources:

Definition: An array in Java is a group of like-typed variables referred to by a common name. Unlike C/C++, arrays in Java are dynamically allocated and stored in contiguous memory locations. Arrays in Java are objects, allowing the use of the object property length to find their length.

Declaration and Initialization: Arrays in Java are index-based, starting at 0. To declare an array, you specify the variable type followed by square brackets. Arrays can be initialized with values in a comma-separated list inside curly braces. For example, int[] myNum = {10, 20, 30, 40} initializes an array of integers.

Accessing Elements: Array elements are accessed by referring to their index number. The first element is at index 0, the second at index 1, and so on. For instance, System.out.println(cars) would output the first element of the cars array.

Changing Elements: To change the value of a specific element in an array, you refer to its index number and assign a new value. For example, cars = "Opel" changes the first element of the cars array to "Opel".

Array Length: The length of an array in Java can be obtained using the length property. For instance, System.out.println(cars.length) would output the number of elements in the cars array.

10.11.1 Types of Arrays

Java supports single-dimensional and multidimensional arrays. Single-dimensional arrays store elements in a linear fashion, while multidimensional arrays store data in row and column-based indexes. Java also allows for jagged arrays, which have varying column sizes in a 2D array.

In Java, there are two main types of arrays: single-dimensional arrays and multi-dimensional arrays.

Single-Dimensional Arrays

Single-dimensional arrays are the most common type of array in Java. They are arrays that have only one dimension, meaning they can store elements in a single row or column. Each element in the array is accessed using an index, which starts from 0.

Multi-Dimensional Arrays

Multi-dimensional arrays are arrays that have more than one dimension. They can be 2D, 3D, or even nD. Each element in a multi-dimensional array is accessed using multiple indices, which are separated by commas. For example, in a 2D array, you would use two indices to access an element, like array$$i]$$j].

Example of Single-Dimensional Array

```
int[ ] myArray = {10, 20, 30, 40};
```

Example of Multi-Dimensional Array

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

10.12 Inheritance

Inheritance in Java is a cornerstone concept of object-oriented programming (OOP) that enables a class to inherit attributes and behaviors from another class. This mechanism fosters

code reusability and establishes a hierarchical relationship among classes. In Java, inheritance is implemented using the extends keyword. For example:

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited from Animal
        dog.bark(); // Defined in Dog
    }
}
```

10.12.1 Types of Inheritance in Java

Java supports several types of inheritance:

Single Inheritance:

A class inherits from only one superclass.

Example

```
class A {
   // Class A code
}
```

```
class B extends A {
   // Class B code
}
```

Multilevel Inheritance:

A class is inherited from another class, which in turn is inherited from yet another class.

Example:

```
class A {
   // Class A code
}

class B extends A {
   // Class B code
}

class C extends B {
   // Class C code
}
```

Hierarchical Inheritance:

Multiple classes inherit from a single superclass.

Example:

```
class A {
   // Class A code
}

class B extends A {
   // Class B code
}

class C extends A {
   // Class C code
}
```

Note on Multiple Inheritance

Java does not directly support multiple inheritance of classes, meaning a class cannot inherit from more than one class, to prevent the complexity and ambiguity of the "Diamond

Problem." However, multiple inheritance can be achieved using interfaces. An interface in Java is an abstract type that defines behaviors that classes must implement.

Example of multiple inheritance using interfaces:

```
interface InterfaceA {
   void methodA();
}
interface InterfaceB {
   void methodB();
}
class MyClass implements InterfaceA, InterfaceB {
   public void methodA() {
      System.out.println("Method A");
   }
   public void methodB() {
      System.out.println("Method B");
   }
}
public class Main {
   public static void main(String[] args) {
      MyClass obj = new MyClass();
      obj.methodA();
      obj.methodB();
   }
}
```

Key Points of Inheritance

super keyword: Used to refer to the immediate parent class object. It is used to call superclass methods and constructors.

Method Overriding: A subclass can offer a specific implementation of a method that is already defined in its superclass.

Protected Members: Members declared as protected in the superclass can be accessed in the subclass.

Constructor Invocation: A subclass constructor can invoke a superclass constructor using the super keyword.

Example of Method Overriding

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.sound();  // Outputs: Dog barks
    }
}
```

In this example, the Dog class overrides the sound method of the Animal class. When the sound method is called on an instance of Dog, the overridden method in Dog is executed. Inheritance is a powerful feature in Java that enables code reuse, polymorphism, and a more natural way to model real-worldrelationships among objects.

10.13 Packages in Java

In Java, a package is a namespace that groups a set of related classes and interfaces. You can think of packages as folders in a file system. Classes in the same package can interact more freely with each other and can help avoid naming conflicts. Packages also provide access protection and ease in maintaining the code.

10.13.1 Creating and Using Packages

Creating a Package:

To create a package, place the `package` keyword followed by the package name at the top of your Java file.

Example:

```
package com.example.myapp;
public class MyClass {
   // Class code here
}
```

Using Packages:

To use a class from another package, you need to import it using the import statement.

Example:

```
import com.example.myapp.MyClass;


public class Main {
   public static void main(String[] args) {
      MyClass myObject = new MyClass();
      // Use myObject
   }
}
```

10.13.2 Default Package:

If a package is not specified, the class falls under the default package. However, in larger projects, it is generally advised to avoid using the default package.

Access Modifiers:

Packages also help in controlling access to classes and members. The public, protected, default (no modifier), and private access levels determine the visibility of classes and their members across different packages.

Example of Package Structure

Consider the following directory structure:

src

```
└── com
    └── example
        ├── MyClass.java
        └── utilities
            └── UtilityClass.java
```

MyClass.java:

```
package com.example;

import com.example.utilities.UtilityClass;

public class MyClass {
    public static void main(String[] args) {
        UtilityClass util = new UtilityClass();
        util.doSomething();
    }
}
```

UtilityClass.java:

```
package com.example.utilities;

public class UtilityClass {
    public void doSomething() {
        System.out.println("Utility class doing something.");
    }
}
```

10.14 Interfaces in Java

Interfaces in Java provide a way to achieve abstraction and multiple inheritance of type. They define a contract for classes to follow by specifying a set of methods that implementing classes must provide. Here's a breakdown of interfaces in Java:

Defining and Implementing an Interface

Defining an Interface:

Use the interface keyword to declare an interface.

Example

```
interface Animal {
```

```
    void eat();
    void sleep();
}
```

10.14.1 Implementing an Interface:

To implement an interface in Java, a class uses the `implements` keyword and must provide concrete implementations for all methods declared in the interface.

Example:

```
class Dog implements Animal {
    public void eat() {
        System.out.println("Dog eats");
    }
    public void sleep() {
        System.out.println("Dog sleeps");
    }
}
```

10.14.2 Using an Interface:

You can create a reference of the interface type and point it to an object of a class that implements the interface.

Example

```
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.eat();
        myDog.sleep();
    }
}
```

10.14.3 Interface Features

Multiple Inheritance:

A class can implement multiple interfaces, thus achieving multiple inheritance.

Example:

```
interface Animal {
    void eat();
}

interface Pet {
    void play();
}

class Dog implements Animal, Pet {
    public void eat() {
        System.out.println("Dog eats");
    }

    public void play() {
        System.out.println("Dog plays");
    }
}
```

Default Methods:

Interfaces can have default methods, which are methods with a body. These provide a default implementation that can be overridden by implementing classes.

Example:

```
interface Animal {
    void eat();

    default void sleep() {
        System.out.println("Animal sleeps");
    }
}
```

```
class Dog implements Animal {
    public void eat() {
        System.out.println("Dog eats");
    }
    // No need to override sleep() unless specific behavior is needed
}
```

Static Methods:

Interfaces can also have static methods.

Example:

```
interface Animal {
    void eat();

    static void description() {
        System.out.println("Animals are living beings.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal.description();  // Calling the static method
    }
}
```

10.15 Exception handling in Java

Exception handling in Java is a powerful mechanism that enables programmers to handle errors or exceptional conditions gracefully. It allows the program to respond to unexpected situations without terminating abruptly. Here's an overview of exception handling in Java:

In Java, exceptions are managed using five keywords: try, catch, finally, throw, and throws.

try:

The try block contains code that might throw an exception. If an exception occurs within the try block, it is handled by the corresponding catch block.

```
try {
    // Code that may throw an exception
```

}

catch:

The catch block is used to handle the exception. It must be immediately after the try block.

```
catch (ExceptionType e) {
   // Code to handle the exception
}
```

finally:

The finally block contains code that is always executed, regardless of whether an exception is thrown or not. It is used to clean up resources like closing files or releasing resources.

```
finally {
   // Code that will always run
}
```

throw:

The throw keyword is used to explicitly throw an exception

```
throw new ExceptionType("Error message");
```

throw new ExceptionType("Error message");

throws:

The throws keyword is used in method signatures to declare the exceptions that a method can throw.

```
public void myMethod() throws ExceptionType {
   // Method code
}
```

10.15.1 Types of Exceptions

Java exceptions are divided into two main categories:

Checked Exceptions:

These are exceptions that are checked at compile time. The programmer is forced to handle these exceptions.

Examples: IOException, SQLException

```
import java.io.*;
public class Example {
   public void readFile() throws IOException {
      FileReader file = new FileReader("test.txt");
      BufferedReader fileInput = new BufferedReader(file);
```

```
        fileInput.close();
    }
}
```

Unchecked Exceptions:

These are exceptions that are not checked at compile time but occur at runtime. They are subclasses of RuntimeException.

Examples: NullPointerException, ArrayIndexOutOfBoundsException

```
public class Example {
    public void divide(int a, int b) {
        int result = a / b; // May throw ArithmeticException
    }
}
```

Example of Exception Handling

```
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]); // This will throw ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of bounds: " + e.getMessage());
        } finally {
            System.out.println("This will always be printed.");
        }

        System.out.println("Rest of the code...");
    }
}
```

Custom Exceptions

Java allows you to create your own exception classes by extending the Exception class or any of its subclasses. Custom exceptions can be useful for application-specific error handling.

10.16 Multithreading in Java

Multithreading in Java is a feature that allows concurrent execution of two or more threads for maximum utilization of CPU. Each thread runs in parallel to perform tasks more efficiently and make the program responsive. Java supports multithreading with the help of the `java.lang.Thread` class and the `java.util.concurrent` package.

A thread is a lightweight process, and Java provides two main ways to create a thread:

By extending the Thread class.

By implementing the Runnable interface.

1. Extending the Thread Class

You can create a new thread by creating a class that extends the Thread class and overriding its run method.

```
class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread: " + i);
            try {
                Thread.sleep(1000); // Pauses the thread for 1000 milliseconds
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // Starts the thread
    }
}
```

2. Implementing the Runnable Interface

You can create a new thread by implementing the Runnable interface and passing an instance of the class to a Thread object.

```
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Runnable: " + i);
            try {
                Thread.sleep(1000); // Pauses the thread for 1000 milliseconds
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start(); // Starts the thread
    }
}
```

10.16.1 Thread Lifecycle

The lifecycle of a thread in Java represents the various states a thread goes through from its creation until its termination. Understanding the thread lifecycle is crucial for effective multithreading programming. Here's an overview of the different states in the thread lifecycle:

Thread States:

1. **New**:
    - When a thread is created but not yet started, it is in the new state.
    - In this state, the thread has been instantiated but has not yet been started via the start() method.
2. **Runnable**:
    - Once the start() method is invoked, the thread enters the runnable state.
    - In this state, the thread is eligible to run but may not be currently executing due to the CPU scheduler.
3. **Blocked (or Waiting)**:

- A thread enters the blocked state when it is temporarily inactive.
- This can occur when the thread is waiting for a monitor lock (synchronized block) or waiting indefinitely for some external event to occur.
- For example, a thread might enter the blocked state while waiting for I/O operations to complete.

4. **Timed Waiting**:
   - Threads can enter the timed waiting state when they invoke methods that cause them to wait for a specific amount of time.
   - Examples of such methods include sleep(), wait(timeout), or join(timeout).
5. **Waiting**:
   - Threads enter the waiting state when they are waiting indefinitely for another thread to perform a particular action.
   - This can happen when a thread invokes the wait() method without a timeout or when it is suspended by another thread.
6. **Terminated (or Dead)**:
   - A thread enters the terminated state when its run() method completes or when an uncaught exception terminates the thread.
   - Once terminated, a thread cannot be restarted.

10.16.2 Thread Synchronization

Synchronization is used to control the access of multiple threads to shared resources. Java provides several ways to synchronize threads:

Synchronized Methods: Synchronize an entire method to ensure that only one thread can execute it at a time.

```
class Counter {
   private int count = 0;
   public synchronized void increment() {
      count++;
   }
   public int getCount() {
      return count;
   }
}
```

Synchronized Blocks: Synchronize a block of code within a method.

```
class Counter {
    private int count = 0;
    private final Object lock = new Object();
    public void increment() {
        synchronized (lock) {
            count++;
        }
    }
    public int getCount() {
        return count;
    }
}
```

10.16.3 Inter-Thread Communication

Inter-thread communication is a fundamental concept in multithreading programming, allowing threads to synchronize their actions and exchange data efficiently. In Java, inter-thread communication is typically achieved using methods like wait(), notify(), and notifyAll() provided by the Object class. Here's an overview of inter-thread communication in Java

```
class SharedResource {
    private int data;
    private boolean available = false;
    public synchronized void produce(int value) {
        while (available) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        data = value;
        available = true;
        notify();
    }
```

```
    public synchronized int consume() {
        while (!available) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        available = false;
        notify();
        return data;
    }
}
public class ProducerConsumerExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        Thread producer = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                resource.produce(i);
                System.out.println("Produced: " + i);
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        Thread consumer = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                int value = resource.consume();
                System.out.println("Consumed: " + value);
                try {
```

```
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    producer.start();
    consumer.start();
  }
}
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExample {
  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(5);

    for (int i = 0; i < 10; i++) {
      Runnable worker = new WorkerThread("" + i);
      executor.execute(worker);
    }
    executor.shutdown();
    while (!executor.isTerminated()) {
    }
    System.out.println("Finished all threads");
  }
}

class WorkerThread implements Runnable {
  private String command;

  public WorkerThread(String s) {
    this.command = s;
  }
```

```
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " Start. Command = " +
command);
        processCommand();
        System.out.println(Thread.currentThread().getName() + " End.");
    }

    private void processCommand() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Concurrency Utilities: The java.util.concurrent package contains several classes to help manage concurrent programming, such as CountDownLatch, CyclicBarrier, Semaphore, ConcurrentHashMap, and more.

```
import java.util.concurrent.*;

public class CountDownLatchExample {
    public static void main(String[] args) throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(3);

        for (int i = 0; i < 3; i++) {
            new Thread(new Worker(latch)).start();
        }

        latch.await(); // Main thread waits until all worker threads finish
        System.out.println("All workers are done!");
    }
}
```

```
class Worker implements Runnable {
    private CountDownLatch latch;

    public Worker(CountDownLatch latch) {
        this.latch = latch;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " is working");
            Thread.sleep(2000);
            System.out.println(Thread.currentThread().getName() + " finished working");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            latch.countDown();
        }
    }
}
```

Multithreading in Java is essential for developing efficient, responsive, and high-performance applications. By understanding and utilizing threads, synchronization mechanisms, and concurrency utilities, you can effectively manage parallel processing and improve the performance of your Java programs.

10.17 String input and output in Java

String input and output in Java are fundamental operations that involve reading and writing string data. Java provides several classes and methods to handle string input and output efficiently.

10.17.1 String Input

1. Using Scanner Class

The Scanner class in java.util package is widely used to read input from various input sources like keyboard (standard input), files, etc.

Reading a single line of text:

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter a line of text:");
        String input = scanner.nextLine();
        System.out.println("You entered: " + input);
        scanner.close();
    }
}
```

Reading a single word:

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter a word:");
        String input = scanner.next();
        System.out.println("You entered: " + input);
        scanner.close();
    }
}
```

2. Using BufferedReader Class

The BufferedReader class in the java.io package can be used to read text from an input stream efficiently.

```
import java.io.BufferedReader;
import java.io.IOException;
```

```
import java.io.InputStreamReader;

public class Main {
    public static void main(String[] args) {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter a line of text:");
        try {
            String input = reader.readLine();
            System.out.println("You entered: " + input);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

10.17.2 String Output

1. Using System.out.println

The System.out.println method is used to print text to the console.

```
public class Main {
    public static void main(String[] args) {
        String message = "Hello, World!";
        System.out.println(message);
    }
}
```

2. Using PrintWriter Class

The PrintWriter class in java.io package provides methods to write formatted text to an output stream.

Writing to the console:

```
import java.io.PrintWriter;

public class Main {
    public static void main(String[] args) {
        PrintWriter writer = new PrintWriter(System.out, true);
```

```
        String message = "Hello, World!";
        writer.println(message);
    }
}
```

Writing to a file:

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class Main {
    public static void main(String[] args) {
        try (PrintWriter writer = new PrintWriter(new FileWriter("output.txt"))) {
            String message = "Hello, File!";
            writer.println(message);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Combining String Input and Output

Here's an example that combines both input and output operations:

```
import java.util.Scanner;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter a line of text:");
        String input = scanner.nextLine();

        try (PrintWriter writer = new PrintWriter(new FileWriter("output.txt"))) {
            writer.println("You entered: " + input);
```

```
            System.out.println("Text written to output.txt");
        } catch (IOException e) {
            e.printStackTrace();
        }
        scanner.close();
    }
}
```

Unit: 11

Introduction to Threads

Objective:

- Grasp the fundamentals of threads and processes, understanding the distinction between them.
- Explain the benefits and drawbacks of using multithreading in Java applications.
- Identify the different states a thread can be in (running, waiting, etc.) and how the Java thread lifecycle works.

Structure:

11.1 Introduction to Multithreading

11.2 Multitasking

11.3 Thread

11.1 Introduction to Multithreading

Multithreading is a fundamental concept in computer science that enables concurrent execution of multiple threads within a single process. Threads are independent sequences of instructions that can be executed concurrently by the CPU. Multithreading allows programs to perform multiple tasks simultaneously, improving performance and responsiveness.

In the context of Java, multithreading is particularly powerful due to its built-in support for creating and managing threads. Java provides a rich set of APIs for multithreading, making it easier for developers to create efficient and responsive applications.

One of the key advantages of multithreading is the ability to execute multiple tasks concurrently, thereby utilizing the available resources more effectively. For example, in a web server application, multithreading can be used to handle multiple client requests simultaneously, improving overall throughput and responsiveness.

Multithreading also enables better utilization of modern multi-core processors. By dividing tasks into smaller threads and executing them in parallel, multithreading allows applications to take advantage of the processing power offered by multi-core CPUs.

However, multithreading introduces challenges such as synchronization and thread safety. Since threads share the same memory space, concurrent access to shared data can lead to race conditions and other concurrency issues. Proper synchronization mechanisms such as locks, semaphores, and monitors are essential to ensure thread safety and prevent data corruption.

Java provides built-in support for synchronization through keywords such as
and classes such as Lock and Semaphore. These mechanisms allow developers to control access to shared resources and coordinate the execution of multiple threads.

Overall, multithreading is a powerful technique for improving the performance and responsiveness of applications. With proper design and synchronization techniques, developers can harness the full potential of multithreading to create fast, efficient, and responsive software systems.

11.1.1 Advantages of Java Multithreading

Java multithreading offers several advantages that make it a powerful tool for developing efficient and responsive applications. Here are some of the key advantages:

1. **Concurrency:** Java multithreading allows multiple tasks to execute concurrently within a single application. This concurrency enables applications to perform multiple operations simultaneously, improving overall throughput and responsiveness.

2. **Resource Utilization:** By leveraging multithreading, Java applications can effectively utilize the available system resources, including CPU cores and memory. Multithreading enables better resource utilization by allowing different parts of the application to execute in parallel.
3. **Improved Performance:** Multithreading can significantly improve the performance of Java applications, especially in scenarios where tasks can be executed concurrently. By distributing tasks across multiple threads, applications can achieve better performance and faster execution times.
4. **Responsiveness:** Multithreading enhances the responsiveness of Java applications by allowing them to remain interactive even while performing time-consuming tasks. By executing lengthy operations in separate threads, applications can maintain a responsive user interface and provide a better user experience.
5. **Scalability:** Java multithreading facilitates scalability by enabling applications to scale horizontally across multiple threads. As the workload increases, additional threads can be created to handle the increased demand, allowing applications to scale with the available resources.
6. **Asynchronous Programming:** Multithreading enables asynchronous programming in Java, allowing tasks to execute independently of one another. Asynchronous execution is particularly useful for performing I/O-bound operations, such as network requests or file I/O, without blocking the main thread.
7. **Modular Design:** Multithreading promotes a modular design approach by allowing developers to divide complex tasks into smaller, more manageable units of work. By encapsulating functionality within separate threads, developers can create more modular and maintainable code.

Overall, Java multithreading offers numerous advantages, including improved concurrency, resource utilization, performance, responsiveness, scalability, asynchronous programming, and modular design. By leveraging these advantages, developers can create efficient, responsive, and scalable Java applications that meet the demands of modern computing environments.

11.2 Multitasking

Multitasking is a fundamental concept in computer science and operating systems that allows multiple tasks or processes to run concurrently on a computer system. It enables efficient

utilization of system resources and enhances overall performance. Multitasking can be classified into two main types:

1. **Preemptive Multitasking:** In preemptive multitasking, the operating system allocates a fixed time slice to each task or process, and then switches between them rapidly. If a task exceeds its allotted time slice, the operating system interrupts it and allocates CPU time to another task. This ensures that all tasks receive fair and timely execution, regardless of their priority or complexity. Preemptive multitasking is commonly used in modern operating systems like Windows, Linux, and macOS.
2. **Cooperative Multitasking:** In cooperative multitasking, tasks voluntarily yield control of the CPU to other tasks when they are idle or waiting for input/output operations to complete. Unlike preemptive multitasking, there is no strict time slicing, and tasks rely on each other to cooperate and share CPU time. Cooperative multitasking requires explicit coordination between tasks and is typically less efficient than preemptive multitasking. It was more prevalent in older operating systems like classic Mac OS and early versions of Windows.

Both types of multitasking have their advantages and disadvantages. Preemptive multitasking offers better system responsiveness, fairness, and stability, as the operating system controls task scheduling and resource allocation. However, it may incur higher overhead due to frequent context switches. Cooperative multitasking can be more efficient in certain scenarios where tasks are well-behaved and cooperate effectively. However, it may suffer from stability issues if tasks do not yield control appropriately.

In summary, multitasking is a crucial feature of modern computer systems, enabling concurrent execution of multiple tasks and enhancing system performance. Whether through preemptive or cooperative mechanisms, multitasking allows users to run multiple applications simultaneously, switch between tasks seamlessly, and make efficient use of system resources.

11.3 Thread

In object-oriented programming (OOP), a thread is the smallest unit of execution within a process. Threads allow concurrent execution of multiple tasks within a single process, enabling parallelism and improving system responsiveness. Threads share the same memory space and resources within a process, allowing them to communicate and synchronize with each other efficiently.

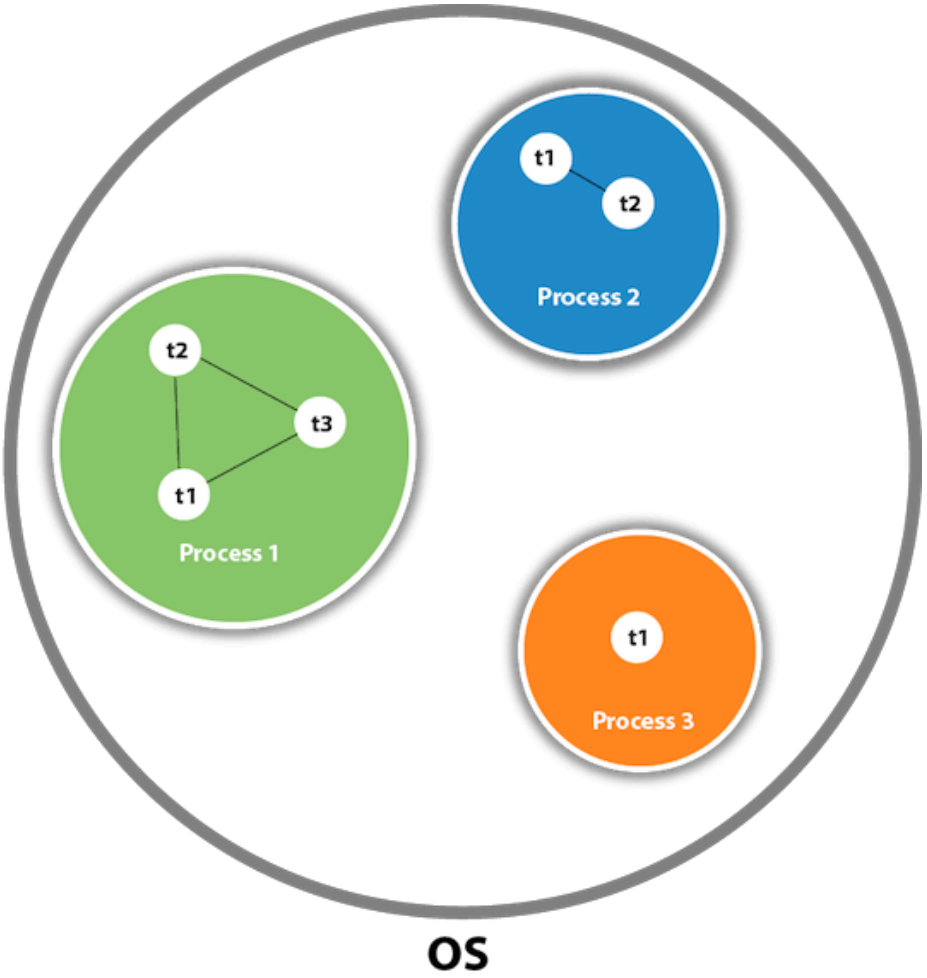


Fig 11.1 Thread

"As shown in the above figure, a thread is executed inside the process". "There is context-switching between the threads". "There can be multiple processes inside the OS, and one process can have multiple threads".

Note: At a time one thread is executed only.

11.3.1 Thread class

In object-oriented programming (OOP), the Thread class is a fundamental component for implementing multithreading. It represents a single thread of execution within a process and provides methods and properties for managing the lifecycle and behavior of the thread.

11.3.2 Life cycle of a Thread (Thread States)

In the realm of multithreading, understanding the life cycle of a thread is crucial. Threads go through several states during their lifetime, and each state represents a different stage of execution. Here's an overview of the life cycle of a thread and its various states:

1. **New:** In this state, a thread is created but has not yet started its execution. The new keyword is used to instantiate a thread object, but the start() method must be called to transition the thread to the runnable state.
2. **Runnable:** Once the start() method is invoked on a thread object, it transitions to the runnable state. In this state, the thread is ready to run, but it may not be currently executing due to the thread scheduler's decisions. When the CPU becomes available, the thread moves to the running state and begins its execution.
3. **Running:** In the running state, the thread is actively executing its code. The thread scheduler allocates CPU time to the thread, allowing it to perform its tasks. Threads in this state may periodically yield the CPU to allow other threads to run, or they may be preempted by higher-priority threads.
4. **Blocked (Waiting/Waiting for Monitor):** Threads can enter the blocked state for various reasons, such as waiting for I/O operations to complete or waiting to acquire a lock on an object's monitor. When a thread is blocked, it temporarily suspends its execution until the condition for which it is waiting is satisfied. Once the condition is met, the thread transitions back to the runnable state and competes for CPU time.
5. **Timed Waiting:** Threads can also enter a special type of blocked state called timed waiting, where they wait for a specified amount of time before resuming their execution. This state is often used when threads need to wait for a particular event to occur within a defined time frame.
6. **Terminated:** The final state in the life cycle of a thread is terminated. Threads enter this state when they have completed their execution or when an error or exception causes them to terminate prematurely. Once a thread is terminated, it cannot be restarted or resumed.

Understanding the life cycle of a thread and its various states is essential for developing multithreaded applications efficiently. By managing thread states effectively and coordinating their execution, developers can create robust and responsive concurrent programs.

11.3.1 Thread class Methods:

Threads are managed through instances of the Thread class, which provides various methods to control and coordinate their execution. Here are some commonly used methods of the Thread class:

1. **start():** This method is used to start the execution of a thread. When called, it invokes the run() method of the thread, causing it to transition from the new state to the runnable state. It should be noted that the start() method can only be called once on a thread; subsequent calls will result in an IllegalThreadStateException.
2. **run():** The run() method contains the code that defines the behavior of the thread. It is automatically called by the JVM when a thread transitions to the runnable state and is scheduled for execution. Developers override this method to define the tasks that the thread should perform.
3. **sleep(long millis):** This method causes the currently executing thread to pause for the specified number of milliseconds. It is often used to introduce delays or to implement timeouts in multithreaded programs. The sleep() method may throw an InterruptedException if the thread is interrupted while sleeping.
4. **join():** The join() method allows one thread to wait for the completion of another thread. When called on a thread object, it causes the current thread to block until the specified thread terminates. This method is useful for coordinating the execution of multiple threads and ensuring that certain tasks are completed before others proceed.
5. **interrupt():** This method interrupts the execution of a thread by setting its interrupt status. When a thread is interrupted, it may respond in different ways depending on how its code is written. For example, a thread performing a blocking I/O operation may throw an InterruptedException when interrupted.
6. **isAlive():** The isAlive() method is used to determine whether a thread is currently active or has terminated. It returns true if the thread is alive (i.e., in the runnable, running, or blocked state) and false if it has terminated.
7. **yield():** The yield() method is a hint to the scheduler that the current thread is willing to yield its current use of the CPU. It allows other threads of the same priority to run, but it does not guarantee that the CPU will be given to another thread immediately.
8. **setName(String name):** This method is used to set the name of a thread. The thread's name can be useful for debugging and monitoring purposes, as it provides a descriptive label for the thread.

These are some of the commonly used methods of the Thread class in Java. By leveraging these methods, developers can effectively control the execution and behavior of threads in multithreaded applications.

11.3.2 Runnable interface:

The Runnable interface provides a way to create threads by implementing a single method called run(). Threads created using the Runnable interface are often preferred over extending the Thread class because Java does not support multiple inheritance, and extending Thread prevents the subclass from extending any other class.

1) Java Thread Example by extending Thread class

```
class Multi extends Thread
{ public void run()
{ System.out.println("thread is running..."); }
public static void main(String args[])
{
Multi t1=new Multi();
t1.start();
}}
```

Output: thread is running...

2) Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi3 m1=new Multi3(); Thread t1 =new Thread(m1); t1.start();
}
}
```

Output: thread is running…

Unit: 12

Introduction to JDBC

Objective:

- Understand the purpose and role of JDBC in enabling Java applications to interact with relational databases.
- Grasp the core components of JDBC architecture, including JDBC driver, DriverManager, Connection, Statement, and ResultSet.
- Explain the advantages of JDBC, such as database and platform independence.

Structure:

12.1 Introduction to JDBC

"JDBC (Java Database Connectivity) is an API that enables Java applications to connect and interact with various relational databases". It provides a standard set of classes and interfaces that developers can use to perform operations like:

- Establishing connections to databases
- Executing SQL statements
- Retrieving and manipulating data
- Handling errors

JDBC offers several advantages, including:

- **Database Independence:** JDBC allows applications to work with different databases without modifying the code, as long as a suitable JDBC driver is available.
- **Platform Independence:** JDBC applications can run on various platforms, including Windows, macOS, and Linux.
- **Security:** JDBC supports features like authentication and authorization to protect database access.

12.2 JDBC Architecture

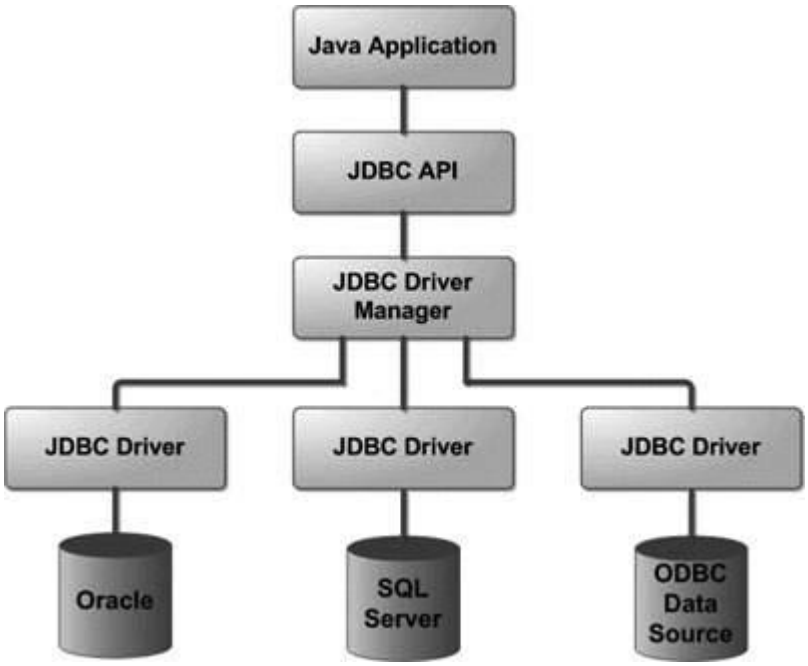


Fig 12.1 JDBC Architecture

JDBC follows a layered architecture consisting of the following components:

- **JDBC Driver:** A specific implementation for a particular database system. It translates JDBC calls into the database system's specific protocol.
- **JDBC DriverManager:** Manages JDBC drivers. It loads the appropriate driver based on the database URL provided in the connection request.
- **Connection:** Represents a session with a database. It allows creation of statements and execution of SQL queries and updates.
- **Statement:** Used to send SQL statements to the database. Different statement types exist for various purposes like queries, updates, and calls.
- **ResultSet:** Represents the result set of a query. It provides methods to navigate through the retrieved data.

12.3 JDBC Connectivity (CODE)

```
 import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class JDBCExample {
    public static void main(String[] args) throws SQLException {
        // Database connection details
        String url = "jdbc:mysql://localhost:3306/your_database";
        String username = "your_username";
        String password = "your_password";
        // Load the JDBC driver
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        // Get a connection
        Connection conn = DriverManager.getConnection(url, username, password);
        System.out.println("Connection established successfully!");
```

```
        // Perform database operations here...
        // Close the connection
        conn.close();
    }
}
```

Connections:

- The application uses the JDBC API to establish a connection with the database through the Driver Manager.
- The Driver Manager locates and loads the appropriate JDBC driver based on the connection URL.
- The loaded driver translates the JDBC API calls into the database's specific protocol.
- Once a connection is established, the application can create statements and execute them on the database.

Benefits of this Architecture:

- **Database Independence:** The application's code remains mostly unchanged regardless of the underlying database, as long as a compatible JDBC driver is available.
- **Platform Independence:** JDBC applications can run on various platforms due to Java's portability.
- **Vendor Flexibility:** You can switch between databases by simply using a different JDBC driver.

12.4 Components in JDBC architecture:

1. JDBC Driver:

- This acts as the translator between the JDBC API and a specific database system.expand_more
- Each database vendor (like MySQL or Oracle) provides its own JDBC driver.expand_more
- The driver translates the calls made using the JDBC API into commands the particular database understands.expand_more

2. JDBC Driver Manager:

- This class manages the collection of loaded JDBC drivers.expand_more
- When your application requests a connection to a database, it provides the connection URL.
- The DriverManager uses this URL to identify the appropriate driver and loads it if necessary.expand_more
- Think of it as a dispatcher that finds the right translator based on the database being accessed.

3. Connection:

- Represents an active session between your Java application and the database.
- Once established, the connection allows you to create statements and execute SQL queries or updates.expand_more
- It's important to close connections after use to release resources.expand_more

4. Statement:

- An object used to send SQL statements (queries, updates, etc.) to the database through the connection. expand_more
- There are different types of statements for various purposes:
    - **Statement:** Used for general purpose SQL statements. expand_more
    - **Prepared Statement:** Pre-compiled statements for improved performance and security against SQL injection attacks.expand_more
    - **Callable Statement:** Used for calling stored procedures in the database. expand_more

5. ResultSet:

- Represents the set of results retrieved from a database query.expand_more
- It provides methods to navigate through the data row by row and access individual columns within each row.expand_more
- You iterate through the ResultSet to process the retrieved data.expand_more

Additional Component:

- **SQLException:** This class represents exceptions that can occur during JDBC operations.expand_more It's essential to handle these exceptions properly in your code.

By understanding these core components, you can effectively interact with various databases using JDBC in your Java applications.

12.5 JDBC Packages:

JDBC utilizes two main packages:

1. **java.sql:** This core package contains the fundamental classes and interfaces for interacting with databases through JDBC. Here are some key elements within this package:
   - **Connection:** Represents a session with a database.
   - **Statement:** Used to send SQL statements to the database (various types exist for different purposes).
   - **ResultSet:** Represents the results of a database query and provides methods to navigate and access the data.
   - **DriverManager:** Manages JDBC drivers and loads the appropriate one based on the connection URL.
   - **SQLException:** Represents exceptions that can occur during JDBC operations.
2. **javax.sql:** This package provides additional features for working with databases in Java applications. It's not always essential but offers functionalities like:
   - **DataSource:** An alternative way to obtain connections from a connection pool, improving performance and connection management.
   - **RowSet:** An alternative to ResultSet for managing data sets.

Remember, when you download the Java Platform Standard Edition (Java SE), you automatically get both these packages included.

REFERANCES

Books:

1. "Object-Oriented Programming in C++" by Robert Lafore
2. "Object-Oriented Software Engineering: A Use Case Driven Approach" by Ivar Jacobson, Grady Booch, and James Rumbaugh
3. "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
4. "Object-Oriented Analysis and Design with Applications" by Grady Booch
5. "Programming: Principles and Practice Using C++" by Bjarne Stroustrup.

Papers:

1. "Design Principles and Design Patterns" by Robert Martin (commonly referred to as the "SOLID principles")
2. "Design Patterns: Abstraction and Reuse of Object-Oriented Design" by Erich Gamma and Richard Helm (the seminal paper introducing design patterns)
3. "A Critique of the Object-Oriented Paradigm" by Christopher Alexander (provides insights into the principles of OOP)
4. "Object-Oriented Programming: An Evolutionary Approach" by Brad J. Cox (introduces the concept of Objective-C and its object-oriented features)
5. "Designing Reusable Classes" by Ralph E. Johnson and Brian Foote (discusses techniques for designing reusable object-oriented software)