Bachelor of Computer Applications (BCA)

OOPS Using C++ (DBCACO302T24)

Self-Learning Material (SEM-III)

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

Jaipur National University

Course Code: DBCACO302T24
Object Oriented Programming Using C++

TABLE OF CONTENTS

Course Introduction	i
Unit 1 Principle of OOP's	1-16
Unit 2 Basics of C++	17-30
Unit 3 Expression	31-41
Unit 4 Function in C++	42-51
Unit 5 Classes and Objects	52-66
Unit 6 Constructor and Destructor	67-77
Unit 7 Operator Overloading and Type Conversion	78-93
Unit 8 Inheritance	94-107
Unit 9 The C++ I/O System Basics	108-128
Unit 10 Working with Files	123–165
Unit 11 Template	151-167
Unit 12 Exception Handling	168-180
Unit 13 Introduction to Standard Template library	181-188
Unit 14 Namespace	189-197
Unit 15 New Style Caste and RTRI	198-206

EXPERT COMMITTEE

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

Mr. Deepak Shekhawat
(Computer and Systems Sciences, JNU Jaipur)

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

COURSE COORDINATOR

Mr. Pawan Jakhar
(Computer and Systems Sciences, JNU Jaipur)

UNIT PREPARATION

Unit Writer(s)	Assisting & Proofreading	Unit Editor
Mr. Pawan Jakhar (Computer and Systems Sciences, JNU Jaipur) (Unit 1-5)	Mr. Hitendra Agarwal (Computer and Systems Sciences, JNU Jaipur)	Dr. Satish Pandey (Computer and Systems Sciences, JNU Jaipur)
Mr. Satender Singh (Computer and Systems Sciences, JNU Jaipur) (Unit 6-10)		
Ms. Komal Sharma (Computer and Systems Sciences, JNU Jaipur) (Unit 11-15)		

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) has revolutionized software development by focusing on objects and classes to create scalable and maintainable applications. This course explores the key OOP principles and their implementation in C++, providing students with a comprehensive understanding of modern programming practices.

This course has 3 credits and is divided into 15 Units. The primary objectives of this course are to introduce students to core OOP concepts such as abstraction, encapsulation, inheritance, and polymorphism. Students will build a strong foundation in C++, learning to leverage its features for various programming tasks. Through practical exercises and projects, students will gain experience in flow control, functions, dynamic memory management, file handling, and exception handling.

The course begins with an overview of different paradigms for problem-solving, highlighting the differences between OOP and procedure-oriented programming. Students will explore the principles of abstraction and the foundational concepts of OOP, including encapsulation, inheritance, and polymorphism. The basics of C++ are introduced, covering the structure of a C++ program, data types, variable declaration, expressions, operators, operator precedence, evaluation of expressions, type conversions, pointers, arrays, and strings.

Flow control statements are essential for directing the flow of a program. This course covers the use of if, switch, while, for, do, break, continue, and goto statements. Functions are another crucial aspect, and students will learn about the scope of variables, parameter passing, default arguments, inline functions, recursive functions, and pointers to functions. The course also covers dynamic memory allocation and deallocation using the new and delete operators, as well as preprocessor directives.

The course then delves into C++ classes and data abstraction. Students will learn about class definitions, class structure, class objects, class scope, the pointer, friends to a class, static class members, constant member functions, constructors, and destructors. Polymorphism is explored in-depth, covering function overloading, operator overloading, and generic programming. The necessity of templates is discussed, along with function templates and class templates.

Inheritance is another critical concept covered in this course. Students will learn how to define a class hierarchy and explore different forms of inheritance. The course covers defining base and derived classes, accessing base class members, and base and derived class construction. Destructors and virtual base classes are also discussed.

Virtual functions and polymorphism are key topics in this course. Students will learn about static and dynamic bindings, base and derived class virtual functions, dynamic binding through virtual functions, the virtual function call mechanism, and pure virtual functions.

Course Outcomes:

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

1. Understand the concept of input and output devices of Computers and how it works and recognize the basic terminology used in computer programming
2. Illustrate the concept of compile and debug programs in C language and use different data types for writing the programs.
3. Design programs connecting decision structures, loops and functions.
4. Distinguish between call by value and call by address.
5. Understand the dynamic behavior of memory by the use of pointers.
6. Use different data structures and create / manipulate basic data files and developing applications for real world problems.

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.

Chapter 1

Principle of OOP's

Objective:

At the end of this session students shall be learning:

- Introduction of Principle of OOP's concept
- Understand the Object Oriented Methodology
- Overview of procedure Oriented programming
- What is 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”

1.1 Software Crisis

The field of software technology is characterized by rapid and continuous advancements, with new tools and techniques being introduced frequently. This dynamic environment necessitates software engineers and the industry to constantly innovate in software design and development practices. This is crucial due to the escalating complexity of software systems and the intense competitiveness within the industry. However, these swift developments have also precipitated a sense of crisis, prompting the need to address several critical issues:

- Representing real-life problems entities in system design.
- Designing system with open interfaces.
- Ensure reusability and extensibility of modules.
- Way to develop modules that are tolerant of any changes in future.
- Improve software productivity and decrease software cost.
- Way to improve quality of software.
- Manage time schedules.

1.2 Software Evaluation

Ernest Tello, a prominent writer in artificial intelligence, likened the evolution of software technology to the growth of a tree. He described distinct phases or "layers" of growth that have accumulated over the past five decades, each layer representing an improvement over its predecessor. Unlike trees, where only the uppermost layer remains functional as the tree grows, in software systems, each layer continues to be operational. This analogy highlights the perpetual functionality and cumulative nature of software technology layers, which build upon each other to enhance capabilities and adapt to evolving technological needs.

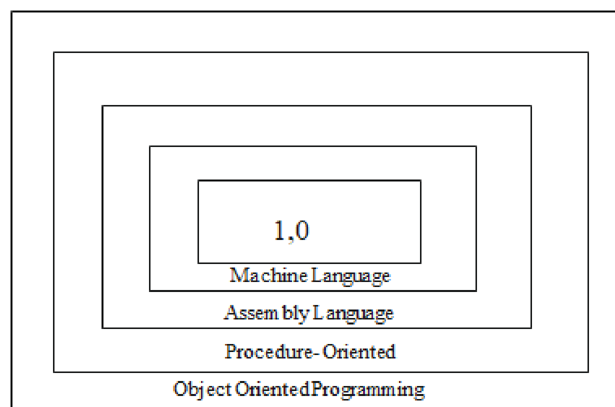


Fig.1.1. Software Layers of Growth

Alan Kay, a key figure in promoting the object-oriented paradigm and a principal designer of Smalltalk, emphasized the importance of architecture as software complexity increases. He stated, "As complexity increases, architecture dominates the basic materials." This underscores the necessity for advanced construction techniques and program structures that are not merely sequences of programming statements or sets of procedures and modules. Instead, they must be designed for ease of comprehension, implementation, and modification.

In the 1980s, structured programming gained popularity with languages like C. It provided a robust methodology for writing moderately complex programs efficiently. However, as software systems grew larger, structured programming alone often fell short in delivering bug-free, maintainable, and reusable code.

Object-Oriented Programming (OOP) emerged as an approach to address these shortcomings by integrating the strengths of structured programming while introducing powerful new concepts. OOP aims to organize and develop programs in a way that enhances maintainability, reusability, and scalability. Unlike a programming language, OOP is a methodology applicable across various programming languages, although not all languages support OOP concepts equally well.

By adopting Object-Oriented Programming (OOP) principles, developers can address many of the challenges posed by complex software systems. OOP encourages modular design, encapsulation, inheritance, and polymorphism, fostering clearer program structures and facilitating easier adaptation to changing requirements over time. This approach aligns with Kay's assertion that effective software architecture is crucial for managing and harnessing the complexities of modern software development.

OOP is a programming paradigm centered on the concepts of classes and objects, and it includes several key principles such as inheritance, polymorphism, abstraction, and encapsulation. These concepts work together to promote code reusability, scalability, and maintainability, making it easier to develop and manage large-scale software projects (Figure 1.2).

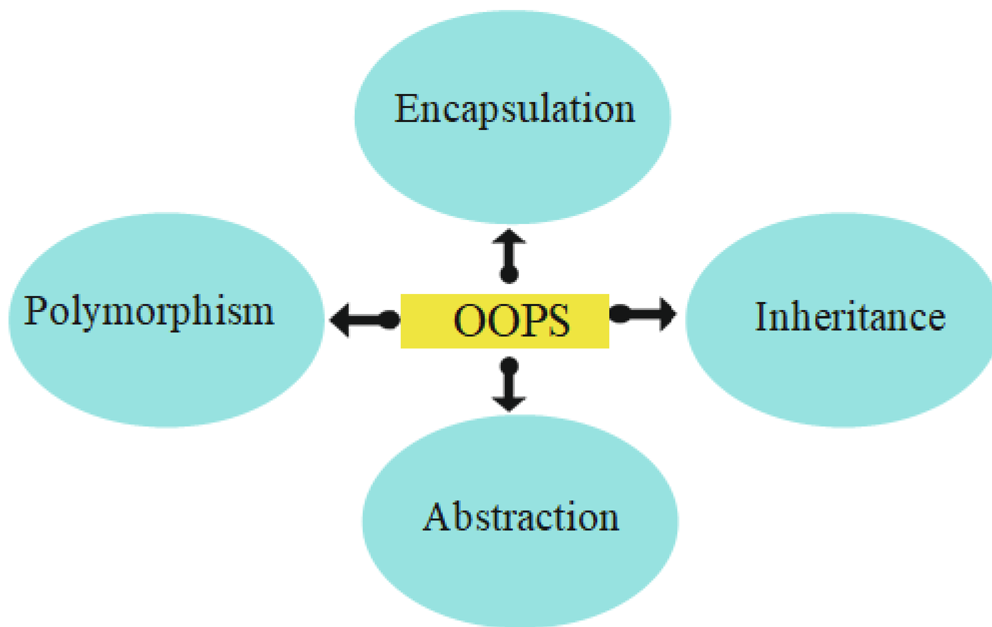


Fig.1.2: OOP's Module

1.3 Procedure-Oriented Programming

In the procedure-oriented approach, problems are tackled as a series of tasks to be completed, such as reading data, performing calculations, and printing results. Languages like COBOL, Fortran, and C exemplify this approach, where the primary focus is on functions. A typical structure for procedural programming involves breaking down a problem into smaller, manageable tasks using hierarchical decomposition. This technique specifies the various tasks needed to solve a problem in a step-by-step manner, ensuring that each function handles a specific aspect of the overall process (Figure 1.3).

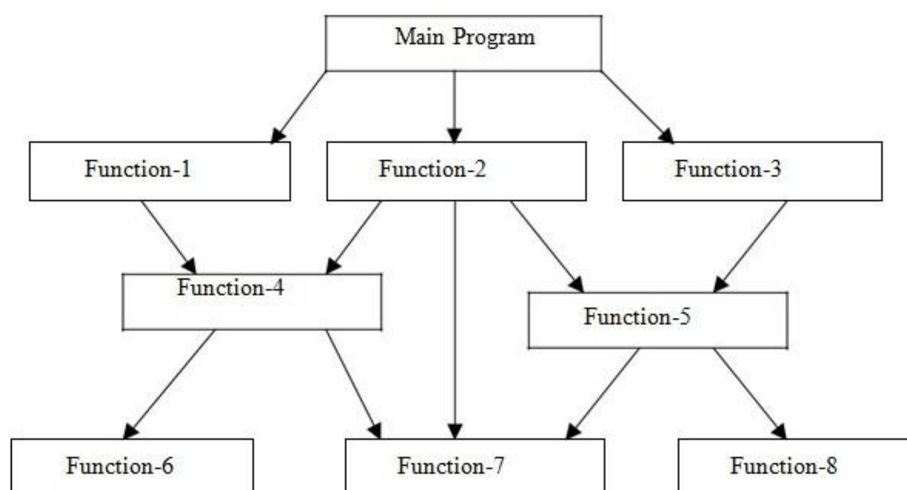


Fig. 1.3 Typical structure of procedural oriented programs

Procedure-oriented programming (POP) involves writing a sequence of instructions for

computer to follow and organizing these instructions into groups called "functions". Typically, flowcharts are used to map out these actions and depict flow of control from one step to next.

In programs with multiple functions, many critical data items are often designated as global so they can be accessed by all functions. While each function can have its own "local data", "global data" are susceptible to unintended changes by any function. In large programs, it becomes challenging to track which data is used by which function. If an external data structure needs modification, all functions that access this data must also be revised, increasing the risk of introducing bugs.

Another significant limitation of the procedural approach is its inadequacy in modeling real-world problems effectively. Functions in POP are action-oriented and do not naturally correspond to real-world entities. This disconnect makes it harder to represent complex problems accurately and intuitively, as the procedural approach focuses more on the actions rather than the elements involved in the problem.

Procedure-oriented programming (POP) exhibits several key characteristics:

- **Emphasis on Algorithms:** The primary focus is on performing tasks and executing algorithms.
- **Modular Structure:** Large programs are broken down into smaller, manageable units known as functions.
- **Global Data Sharing:** Many functions share access to global data, allowing for easier data manipulation across different parts of the program.
- **Data Flow:** Data moves openly from one function to another throughout the system.
- **Data Transformation:** Functions are designed to transform data from one form to another as they execute.
- **Top-Down Design:** POP employs a top-down approach in program design, starting from the highest-level task and breaking it down into smaller, more detailed functions.

1.4 Object Oriented Paradigm

The primary motivation behind development of object-oriented approach was to address the shortcomings of procedural paradigm. Object-Oriented Programming (OOP) treats data as a fundamental component of program development, ensuring it is not freely shared across the

system. Instead, OOP tightly couples data with the functions that operate on it, safeguarding it from unintended modifications by external functions.

In OOP, problems are decomposed into distinct entities known as objects. These objects encapsulate both data and the functions that manipulate this data. The organization of data and functions in object-oriented programs is illustrated in Figure 1.4. Within this framework, the data of an object can only be accessed by the functions associated with that specific object. Nevertheless, functions from one object are permitted to interact with the functions of other objects. This encapsulation and modularity enhance the robustness and maintainability of the software (Figure 1.4).

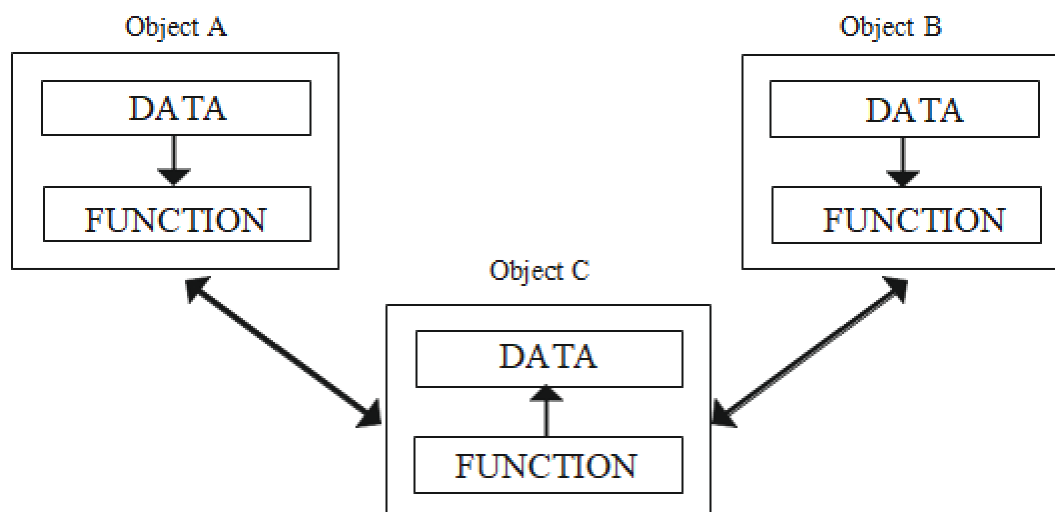


Fig 1.4: Organization of data and function in OOP

Object-oriented programming (OOP) has several distinctive features:

- **Data-Centric Approach:** Emphasis is placed on data rather than procedures.
- **Object Division:** Programs are divided into entities known as objects.
- **Characterized Data Structures:** Data structures are designed to represent the characteristics of objects.
- **Encapsulated Functions:** Functions that operate on an object's data are encapsulated within the data structure.
- **Data Encapsulation:** Data is hidden and cannot be accessed directly by external functions.
- **Object Communication:** Objects can communicate with each other through functions.

- **Ease of Extension:** New data and functions can be added easily as needed.
- **Bottom-Up Design:** OOP follows a bottom-up approach in program design.

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

1.5 Basic Concepts of Object Oriented Programming

It is necessary to understand some of concepts used extensively in object-orinted programming. These include:

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

1.5.1 Objects

Objects are fundamental runtime entities in an object-oriented system. They can represent various real-world and conceptual items, such as a "person, place, bank account, data table, or any item program needs to manage". Objects may also encompass user-defined data types like vectors, time, and lists. The analysis of a programming problem is conducted in terms of objects and communication between them. Ideally, program objects should closely mirror real-world objects.

Objects occupy memory space and have associated addresses, similar to records in Pascal or structures in C. During program execution, objects interact by sending messages to each other. For instance, in a program with "customer" and "account" objects, the customer object might send a message to account object to request bank balance. Each object contains data and the code to manipulate that data. This interaction allows objects to communicate without needing to know the internal details of each other's data or code. It is sufficient to know type of message an object accepts and type of response it returns.

Different authors represent objects in various ways, but Figure 1.5 illustrates two notations commonly used in object-oriented analysis and design.

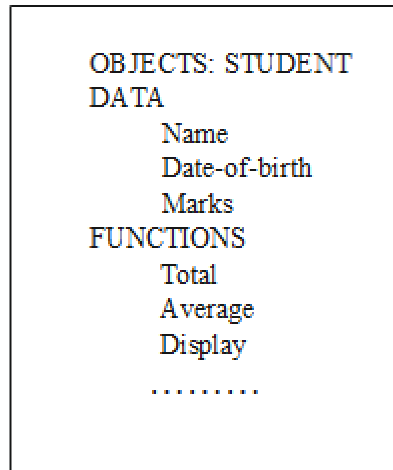


Fig. 1.5 representing an object

1.5.2 Classes

As previously mentioned, objects encapsulate both data and code manipulation functionalities. By employing classes, the entirety of an object's data and code can be encapsulated into a user-defined data type. In essence, objects serve as variables of the class type, allowing for the creation of multiple instances. Each object is linked to the class's data structure upon instantiation. Consequently, a class constitutes a group of objects sharing similar characteristics. For instance, in a class named "Fruit," objects like "Mango," "Apple," and "Orange" can be instantiated. Classes, being user-defined types, mirror the behavior of built-in types within a programming language. The syntax employed to instantiate an object is analogous to creating an integer object in C, facilitating ease of use and comprehension. If fruit has been defines as a class, then the statement:

Fruit Mango;

Will create an object mango belonging to the class fruit.

1.5.3 Data Abstraction and Encapsulation

Encapsulation, the amalgamation of data and functions into a cohesive unit known as a class, is a fundamental principle in programming. It serves as a hallmark feature of classes, ensuring data security and integrity by restricting direct access from the outside world. Only functions encapsulated within the class can access its data, thereby providing a controlled interface between the object's data and the program. This safeguarding of data from external access is commonly referred to as data hiding or information hiding.

Abstraction, on the other hand, entails representing essential features without delving into background details or explanations. Classes leverage abstraction by defining abstract attributes such as size, weight, and cost, along with functions that operate on these attributes. By encapsulating the essential properties of an object, classes facilitate abstraction, enabling developers to focus on core functionalities without getting bogged down by implementation specifics.

The attributes within a class are often referred to as data members, as they hold information pertinent to the object's state. Correspondingly, functions that manipulate these data members are termed methods or member functions, encapsulating the behavior associated with the object's data.

1.5.4 Inheritance

Inheritance, a cornerstone concept in object-oriented programming (OOP), facilitates the acquisition of properties from one class by objects of another class. It underpins the notion of hierarchical classification, allowing for the organization of entities into a structured hierarchy. For instance, a "robin," a type of bird, belongs to the "flying bird" class, which in turn is a subclass of the broader "bird" class. This hierarchical arrangement reflects the shared characteristics among derived classes, as depicted in Figure 1.6.

In OOP, inheritance embodies the principle of reusability, enabling the extension of existing classes without altering their core structure. This is achieved by deriving a new class from an existing one, resulting in the new class inheriting the combined features of both parent and subclass. Thus, inheritance empowers developers to enhance and specialize functionality by building upon the foundation established in parent classes, fostering code modularity and extensibility. The real appeal and power of the inheritance mechanism is that it

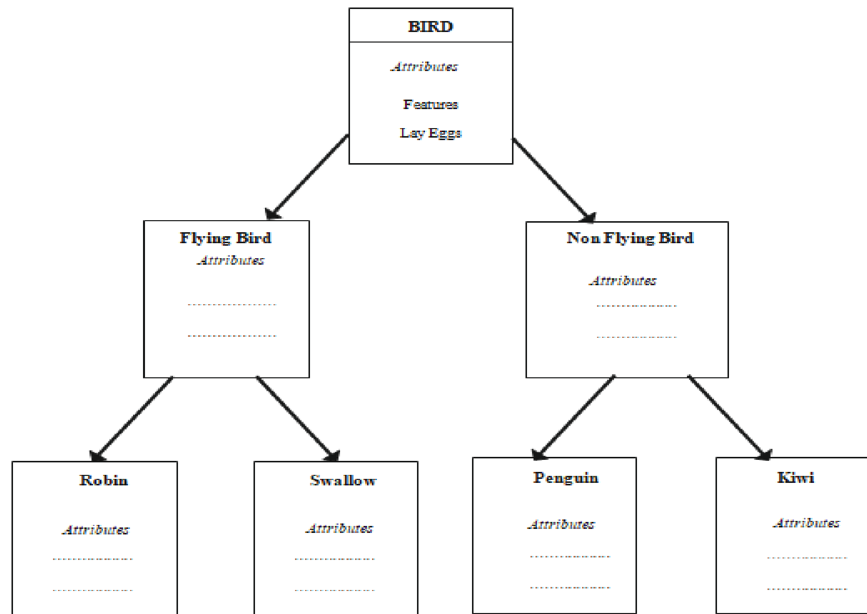


Fig. 1.6 Property inheritances

Allows the programmer to reuse a class i.e. almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduced any undesirable side- effects into the rest of classes (Figure 1.6).

1.5.5 Polymorphism

Polymorphism is a key concept in object-oriented programming (OOP). The term, derived from Greek, means the ability to take on multiple forms. In OOP, polymorphism allows an operation to exhibit different behaviors in different instances, depending on the types of data involved (Figure 1.7).

For example, consider the addition operation. When applied to two numbers, it generates a sum. If the operands are strings, the operation concatenates them to produce a third string. This ability of an operator to exhibit different behaviors based on the context is known as operator overloading.

Similarly, a single function name can handle different numbers and types of arguments, akin to a word having several meanings depending on the context. This is known as function overloading, where one function name can perform various tasks depending on the arguments provided.

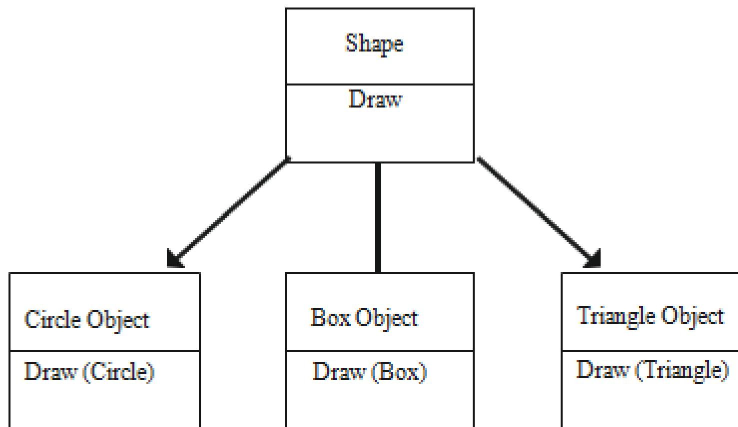


Fig. 1.7 Polymorphism

Polymorphism is crucial in enabling objects with different internal structures to share the same external interface. This allows a general class of operations to be accessed uniformly, even though the specific actions associated with each operation may vary (Figure 1.7).

1.5.6 Dynamic Binding

Binding entails associating a procedure call with the corresponding code to be executed in response to the call. Dynamic binding, a characteristic feature of polymorphism and inheritance, implies that the code linked to a particular procedure call is determined only at runtime. This means that the specific implementation of a function associated with a polymorphic reference is determined by the dynamic type of that reference when the function is called.

Consider the "draw" procedure illustrated in Figure 1.7. Through inheritance, every object inherits this procedure. However, the algorithm executed by the "draw" procedure varies for each object, leading to its redefinition in each class defining the object. During runtime, the code corresponding to the object referenced at that moment is invoked, showcasing the dynamic nature of binding in object-oriented programming.

1.5.7 Message Passing

In object-oriented programming, the process involves several foundational steps:

1. “Creating classes” that define object and their behavior,
2. “Creating objects” from class definitions, and
3. “Establishing communication among objects”.

In object-oriented programming, objects communicate with each other through message passing, which simulates how people exchange information. When an object sends a message to another object, it's essentially requesting the execution of a procedure or method defined within the receiving object. This process invokes the function associated with the message and produces the desired outcome.

For instance, in message passing, you specify:

- The name of the object sending the message.
- The name of the function or message being invoked.
- Any additional information or parameters that need to be sent along with the message.

  This approach allows objects to interact by invoking methods and exchanging data, enabling the construction of systems that closely model or simulate real-world scenarios and interactions (Figure 1.8).

Example:

Employee. Salary (name)

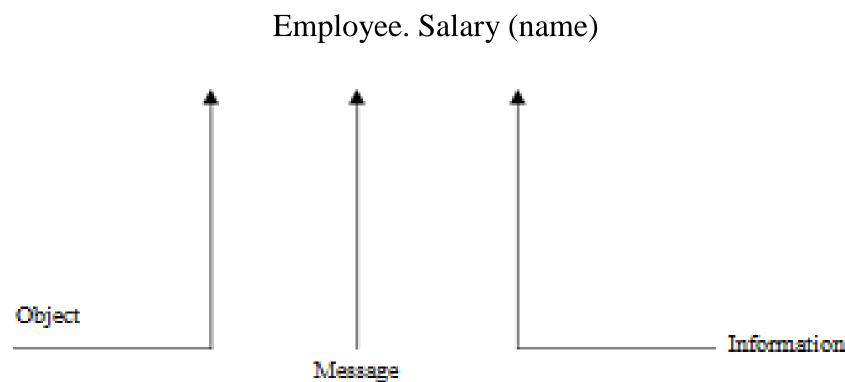


Fig 1.8: Employee Salary

Everything has a life cycle. They are able to be made and destroyed. An object can be communicated with as long as it is living.

1.6 Benefits of OOP

Object-oriented programming (OOP) provides numerous benefits to both program designers and users, addressing many challenges associated with software development and enhancing software quality. Here are the principal advantages of OOP:

- **Code Reusability through Inheritance:** Inheritance allows the elimination of redundant code by extending the use of existing classes. This promotes modular design and facilitates easier maintenance and updates.

- **Modularity and Reusability:** Programs can be constructed using standard working modules (objects) that communicate with each other. This modular approach saves development time, enhances productivity, and improves code organization.
- **Data Hiding for Security:** Encapsulation and data hiding help create secure programs by preventing external code from accessing sensitive data within objects. This ensures better security and reduces the risk of unintended data modification.
- **Support for Multiple Instances:** Objects can exist as multiple instances simultaneously without interfering with each other, promoting flexibility and scalability in program design.
- **Mapping to Problem Domain:** OOP allows developers to map objects directly to entities in the problem domain, facilitating clearer and more intuitive software modeling.
- **Effective Project Management:** Partitioning work based on objects enables efficient project management, with teams focusing on specific modules or objects, leading to better collaboration and code maintenance.
- **Detailed Modeling:** The data-centered design approach in OOP enables capturing intricate details of models and implementing them effectively in code.
- **Scalability:** OOP systems can easily scale from small to large systems, accommodating growth and expansion without significant redesign.
- **Simplified Interface Design:** Message passing between objects simplifies interface descriptions with external systems, enhancing interoperability and integration capabilities.
- **Managing Software Complexity:** OOP provides tools and methodologies to manage software complexity effectively, leading to clearer, more maintainable codebases.

While object-oriented systems offer a range of features that can significantly benefit software projects, their relevance depends on the specific project type and the preferences of the programmer. However, several challenges must be addressed to fully realize these benefits.

For example, the availability of comprehensive object libraries is crucial for effective code reuse. As technology evolves rapidly, current products may quickly become outdated, necessitating strict controls and protocols to ensure that reuse efforts are not compromised.

1.7 Object Oriented Language

Object-oriented programming (OOP) is not confined to specific languages; rather, it can be implemented in various programming languages like C and Pascal. However, as programs grow larger, implementing OOP concepts in such languages can become cumbersome and prone to confusion. Thus, languages specifically designed to support OOP make implementation more straightforward.

Languages claiming to support OOP must encompass several key OOP concepts, leading to their classification into two categories:

1. "Object-based Programming Languages": These languages primarily focus on encapsulation and object identity. Key features include:
   - "Data encapsulation"
   - "Data hiding and access mechanisms"
   - "Automatic initialization and cleanup of objects"
   - "Operator overloading"

   Ada is an example of a typical object-based programming language.

2. Object-oriented Programming Languages: These languages encompass all object-based programming features along with two additional features: inheritance and dynamic binding. Object-oriented programming can thus be characterized by:

   "Object-based features + Inheritance + Dynamic binding"

   These additional features enhance the flexibility and modularity of the codebase, allowing for more sophisticated and scalable software solutions.

1.8 Application of OOP

Object-oriented programming (OOP) has emerged as a prominent trend in modern software engineering, eliciting widespread interest and excitement among software engineers. While its applications were initially prevalent in user interface design, particularly in windowing systems, OOP has now found relevance in a myriad of areas.

Real-world business systems, often characterized by complexity and numerous interconnected

objects with intricate attributes and methods, benefit from OOP's ability to simplify such intricate problems. OOP's promising applications extend to various domains, including real-time systems, simulation and modeling, object-oriented databases, hypertext and hypermedia, AI and expert systems, neural networks and parallel programming, decision support and office automation systems, as well as CIM/CAM/CAD systems.

The evolution of OOP from language to design and now into analysis signifies its maturity and adaptability across different phases of software development. It is anticipated that the comprehensive OOP environment will not only enhance the quality of software systems but also improve productivity within the software industry. Consequently, object-oriented technology is poised to revolutionize the thought processes, analysis techniques, design methodologies, and implementation practices employed by software engineers in developing future systems.

1.9. Summary.

OOP offers several benefits to both the program designer and the user. Object- Orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. Today, traditional programming languages need to be modified and needs the development of Object-Oriented Programming concepts, as it is easily understood by Human being. More things are expected when you are learning the language of C++.

Over the past five decades, software technology has undergone a series of transformations. Procedural-oriented programming (POP) follows a top-down approach, viewing problems as sequences of tasks to be performed, with functions written to execute these tasks. However, POP suffers from two major drawbacks: data can freely move around the program, and it doesn't model real-world problems effectively. Object-oriented programming (OOP) was developed to address these limitations, adopting a bottom-up approach. In OOP, problems are viewed as collections of objects, where objects are instances of classes. Data abstraction involves encapsulating essential features without delving into background details.

Inheritance is a key feature of OOP, allowing objects of one class to acquire properties from objects of another class. Polymorphism enables the use of a single name to represent multiple forms, permitting the existence of multiple functions with the same name in a program.

Dynamic binding ensures that the code associated with a procedure is determined at runtime. Message passing involves specifying the object's name, function's name, and information to be sent.

C++, a superset of the C language, introduces several features such as objects, inheritance, function overloading, and operator overloading to C. It also supports interactive input and output features and introduces the // symbol for single-line comments. Similar to C programs, execution of all C++ programs begins at the main() function. These advancements in C++ enrich programming capabilities, fostering more modular and robust software development.

1.10. Self-Assessment Questions

1. What are the main factors contributing to the software crisis?
2. How does the procedural-oriented programming (POP) paradigm contribute to the challenges faced in software development?
3. What are the limitations of POP in effectively managing complex software projects?
4. How does object-oriented programming (OOP) address the shortcomings of POP in software development?
5. What are the key principles of OOP, and how do they promote better software design and development?
6. Can you provide examples of real-world problems that are better suited for OOP compared to POP?
7. How does OOP improve code reusability and maintainability compared to POP?
8. What role does abstraction play in both POP and OOP, and how does it impact software development practices?
9. What are some of the challenges associated with transitioning from POP to OOP in software projects?

Chapter 2

Basics of C++

Objective:

At the end of this session students shall be learning:

- 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++

2.2. Difference between C and C++

2.3. Features of C++

2.4. Advantages and Disadvantages of C++

2.5. Applications of C++

2.6. Writing and Executing a C++ Program

2.7. Program Structure and Rules

2.8. Sample C++ Program

2.9. Comments

2.10. Return Type of MAIN()Namespace std

2.11. Header File

2.12. Output Statement (COUT)

2.13. Input Statement (CIN)

2.14. Text Editor

2.15. C++ Complier
2.16. Summary
2.17. Self Assessment QuestionsNote

2.1. A Brief History of C and C++

C is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system. C is the most widely used computer language. It keeps fluctuating at number one scale of popularity along with Java programming language, which is also equally popular and most widely used among modern software programmers.

C++, an object-oriented programming language, emerged from the hands of Bjarne Stroustrup at AT&T Bell Laboratories in the early 1980s. Stroustrup, influenced by Simula67 and a dedicated advocate of C, aimed to amalgamate the strengths of both languages, resulting in a more potent language capable of supporting object-oriented programming while preserving C's robustness and elegance. The outcome was C++, an extension of C with the significant addition of the class construct from Simula67. Initially dubbed 'C with classes' due to the pivotal role of classes, the language was officially named C++ in 1983, reflecting its evolutionary nature denoted by the increment operator ++ in C.

C++ serves as a superset of C, with nearly all C programs also qualifying as C++ programs. However, there exist minor discrepancies that may prevent a C program from compiling under a C++ compiler. Notably, C++ introduces vital enhancements such as classes, inheritance, function overloading, and operator overloading. These features empower the creation of abstract data types, inheritance of properties from existing data types, and support for polymorphism, thereby solidifying C++'s status as a truly object-oriented language.

2.2. Difference between C and C++

C and C++ are both programming languages. C is a procedural programming language whereas C++ is an object oriented programming language. There were certain drawbacks in the C language, because of that C++ was developed.

The major difference between C and C++ is that C is a procedural programming language and does not support classes and objects, while C++ is a combination of both procedural and object oriented programming language; therefore C++ can be called a hybrid language.

The following table presents differences between C and C++ in detail.

C	C++
C was developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs.	C++ was developed by Bjarne Stroustrup in 1979 with C++'s predecessor "C with Classes".
When compared to C++, C is a subset of C++.	C++ is a superset of C. C++ can run most of C code while C cannot run C++ code.
C supports procedural programming paradigm for code development.	C++ supports both procedural and object oriented programming paradigms; therefore C++ is also called a hybrid language.

C	C++
C does not support object oriented programming; therefore it has no support for polymorphism, encapsulation, and inheritance.	Being an object oriented programming language C++ supports polymorphism, encapsulation, and inheritance.
In C (because it is a procedural programming language), data and functions are separate and free entities.	In C++ (when it is used as object oriented programming language), data and functions are encapsulated together in form of an object. For creating objects class provides a blueprint of structure of the object.
In C, data are free entities and can be manipulated by outside code. This is because C does not support information hiding.	In C++, Encapsulation hides the data to ensure that data structures and operators are used as intended.
C, being a procedural programming, it is a function driven language.	While, C++, being an object oriented programming, it is an object driven language.
C does not support function and operator overloading.	C++ supports both function and operator overloading.
C does not allow functions to be defined inside structures.	In C++, functions can be used inside a structure.

C does not have namespace feature.	C++ uses NAMESPACE which avoid name collisions. A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries. All identifiers at namespace scope are visible to one another without qualification. Identifiers outside the namespace can access the members by using the fully qualified name for each identifier.
C uses functions for input/output. For example scanf and printf.	C++ uses objects for input output. For example cin and cout.
C does not support reference variables.	C++ supports reference variables.
C has no support for virtual and friend functions.	C++ supports virtual and friend functions.
C provides malloc() and calloc()functions for dynamic memory allocation, and free() for memory de-allocation.	C++ provides new operator for memory allocation and delete operator for memory de-allocation.
C does not provide direct support for error handling (also called exception handling)	C++ provides support for exception handling. Exceptions are used for "hard" errors that make the code incorrect.

2.3. Features of C++

1. Multi-Paradigm: Supports procedural, object-oriented, and generic programming paradigms.
2. Efficiency: Known for performance and direct hardware access, ideal for system-level programming.
3. Standard Library: Rich collection of data structures, algorithms, and utilities for rapid development.
4. Object-Oriented Programming (OOP): Classes, objects, encapsulation, inheritance, and polymorphism for modular and reusable code.
5. Templates: Enables generic programming, enhancing code flexibility and performance.
6. Pointer Support: Allows direct memory manipulation and efficient resource handling.
7. Platform Independence: Code portability across different platforms and architectures.
8. Low-Level Manipulation: Features for low-level programming, like pointer arithmetic.
9. Exception Handling: Mechanisms to gracefully manage errors and exceptions.
10. Operator Overloading: Custom definitions for operators in user-defined types, improving code expressiveness.

2.4. Advantages and Disadvantages of C++

C++, built upon the foundations of the C language, emerged in the early 1980s through the pioneering work of Bjarne Stroustrup at AT&T Bell Laboratories. The choice of the name "C++" reflects its lineage, with the "++" symbol signifying an extension, as it is a syntactic construct in C used for incrementing a variable. Notably, C++ encompasses a significant portion of C's syntax and functionality, rendering it a superset of C. Consequently, the majority of C programs can be compiled using a C++ compiler, underscoring the compatibility and evolutionary relationship between the two languages. A C++ program is a collection of commands, which tell the computer to do "something." This collection of commands is usually called C++ source code

C++ is the Mid-Level programming language because it acquires the feature of Low level as well as high-level programming language. Using C++ Programming Language we can create a different kind of Software, These are:

- System software, application software
- device drivers, embedded software
- high-performance server and client applications and
- entertainment software such as video games
- Object oriented
- Portable language (writing a program irrespective of operating system as well as Hardware)
- Low-level language like Assembly language on Machine language called portable.
- C++ embraces multi-paradigm programming, accommodating various styles of programming. A paradigm dictates the logical, structural, and procedural aspects of a program. C++ is renowned for its support of three primary paradigms: Generic, Imperative, and Object-Oriented.
- This versatility renders C++ invaluable for low-level programming tasks while remaining highly efficient for general-purpose applications
- C++ provide performance and memory efficiency.
- It provides a high-level abstraction.

- In the language of the problem domain.
- C++ is compatible with C.
- C++ used reusability of code.
- C++ used inheritance, polymorphism.
- Portability allows developing program irrespective of hardware
- It allows moving the program development for one platform to another platform

C compatible (COMP) : Programs developed in 'C' language can be moved without any modifications into C++

C++ is an object oriented embedded language which is having the characteristics of low-level language & which is also developing the embedded software. Which language having low-level features

Disadvantages

- It has no security
- Complex in a very large high-level program.
- Used for platform specific application commonly.
- For a particular operating system or platform, the library set has usually chosen that locks.
- When C++ used for web applications complex and difficult to debug.
- C++ can't support garbage collection.
- C++ is not secure because it has a pointer, friend function, and global variable.
- No support for threads built in.

2.5. Applications of C++

Mainly C++ Language is used for Develop Desktop application and system software. Some application of C++ language are given below.

- For Develop Graphical related application like computer and mobile games.
- To evaluate any kind of mathematical equation use C++ language.
- Like window xp , C++ Language are also used for design OS.
- Google also use C++ for Indexing
- Few parts of apple OS X are written in C++ programming language.

- Internet browser Firefox are written in C++ programming language
- All major applications of adobe systems are developed in C++ programming language. Like Photoshop, ImageReady, Illustrator and Adobe Premier.
- Some of the Google applications are also written in C++, including Google file system and Google Chromium.
- C++ are used for design database like MySQL.

2.6. Writing and Executing a C++ Program

Installation of TC is very simple just download turbo C or C++ and run .exe files

When you install the Turbo C or C++ compiler on your system, then TC directory is created on the hard disk and various sub directories such as INCLUDE, and LIB etc. are created under TC (Figure 2.1).

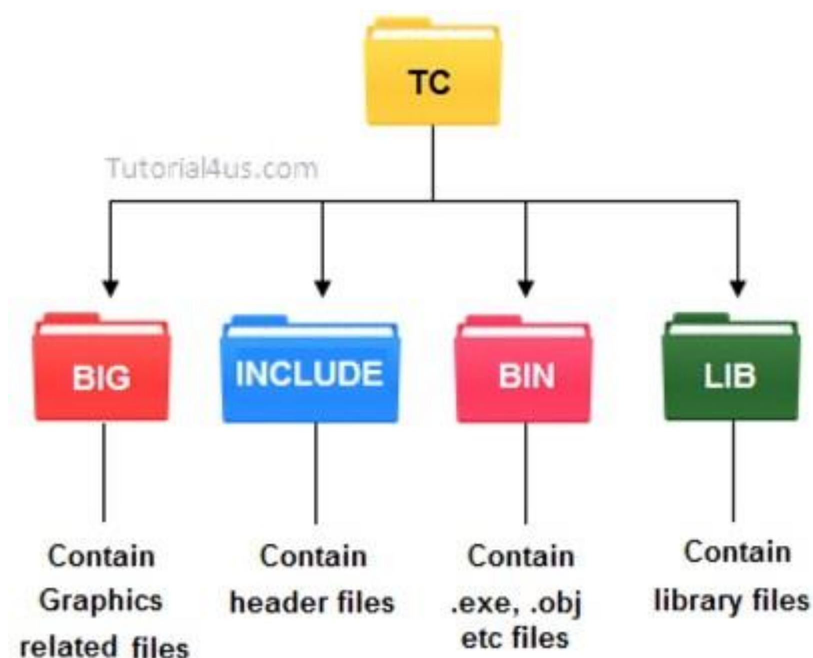


Figure 2.1 Turbo C

- **INCLUDE:** Contain the header files of C and C++.
- **LIB:** Contain the library files of C and C++.
- **BGI:** Contain Graphics related files.
- **BIN:** Contain .exe, .obj etc files.
- **TC Editor:** TC Editor is very simple and easy to use; here i will give you all tips related to TC Editor and some shortcut keys related to TC Editor which is very useful at the time of coding. Turbo C is a most common C language compiler. Below i will

discuss all about its Interfaces.

TC Editor: The interface of Turbo C is very simple. When IDE screen appears, the menu bar is activated. It contains various menus such as:

- **File:** This menu contains group of commands used for save , edit , print program, exit from Turbo C editor etc.
- **Edit:** This menu contains group of commands used for editing C program source code. Example Copy, Cut, Paste, Undo etc.
- **Search:** This menu contains group of commands used for searching specific word as well as replacing it with another one.
- **Run:** This menu contains group of commands used for running C program.
- **Compile:** This menu contains group of commands used for compiling C program.
- **Debug:** This menu contains group of commands used for debugging C program.

Project: This menu contains group of commands used for opening, closing and creating projects.

- **Options:** This menu contains group of commands used for configuring IDE of Turbo C and setting up directories etc.
- **Windows:** This menu contains group of commands used for opening, closing various windows of IDE.
- **Help:** This menu is used to get help about specific topic of C language. Similarly to get help about a specific keyword or identifier of C.

Shortcut keys Related to TC Editor

- Alt + x : Close TC Editor.
- Clt + f9 : Run C Program.
- Alt + f9 : Compile C Code.
- Alt + Enter : Get Full Screen or Half Screen TC Editor.
- Clt + y : Delete complete above line of cursor.
- Shift + Right arrow : Select Line of Code.
- Clt + Insert : Copy.
- Shift + Insert : Paste.
- Shift + Delete : Delete.

Figure 2.2 Turbo C Editor

2.7. Program Structure and Rules

Like C, the C++ program is a collection of function. The above example contain only one function main(). As usual execution begins at main(). Every C++ program must have a main(). C++ is a free form language. With a few exception, the compiler ignore carriage return and white spaces. Like C, the C++ statements terminate with semicolons.

2.8. Sample C++ Program

C++ basic programs in simple and easy way, here I will use C++ programming language for coding of any programs. Also I will discuss all program with the help of pictures and most easy and real life examples. We are trying to give a best and easy tips, tricks and way of programming. I hope this guideline is helpful for all our visitors to learn C++.

The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones.
C++ supports a variety of programming styles. You can write in the style of Fortran, C, Smalltalk, etc., in any language. Each style can achieve its aims effectively while maintaining runtime and space efficiency. C++ is used by hundreds of thousands of programmers in essentially every application domain. C++ is being highly used to write device drivers and other software that rely on direct manipulation of hardware under real-time constraints. C++ is

widely used for teaching and research because it is clean enough for successful teaching of basic concepts. Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are written in C++.

Let us begin with a simple example of a C++ program that prints a string on 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

C++ introduces a new comment symbol // (double slash). Comment start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line, and whatever follows till the end of the line is ignored. Note that there is no closing symbol.

The double slash comment is basically a single line comment. Multiline comments can be written as follows:

“// This is an example of”

“// C++ program to illustrate”

“// some of its features”

The C comment symbols /,/ are still valid and are more suitable for multiline comments. The following comment is allowed:

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

/”

The only statement in program 1.10.1 is an output statement. The statement Cout<<”C++ is better than C.”;

Causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, “cout” and “<<”. The identifier cout(pronounced as C out) is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices. The operator “<<” is called the insertion or put to operator.

2.10. Return Type of MAIN( )

In C++, main () returns an integer value to the operating system. Therefore, every main( ) in C++ should end with a return (0) statement; otherwise a warning an error might occur. Since

main( ) returns an integer type for main () is explicitly specified as int. Note that the default return type for all function in C++ is int. The following main without type and return will run with a warning:

```
main ()
{
…………..

---------
}
```

2.11. Namespace std

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifier defined in the namespace scope we must include the using directive, like

Using namespace std;

Here, std is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global scope. Using and namespace are the new keyword of C++.

2.12. Header File

Header files are included at the beginning, akin to C++ programs. Here, "iostream" serves as a header file providing input and output streams. These files contain pre-declared function libraries for user convenience. In the program, we utilize the #include directive:

#include <iostream>

This directive instructs the compiler to integrate the contents of the file enclosed within angular brackets into the source file. The "iostream" header file should be included at the outset of all programs utilizing input/output statements.

The statement `cin >> number1;` represents an input statement, prompting the program to await user input of a number. The entered number is then stored in the variable "number1". The identifier "cin" (pronounced 'C in') is a predefined object in C++ corresponding to the standard input stream, representing the keyboard.

The ">>" operator, known as the extraction or get-from operator, retrieves the value from the keyboard and assigns it to the variable on its right. This operation resembles the familiar "scanf()" function in C. Similar to "<<" for output, the ">>" operator can also be overloaded (Figure 2.3).

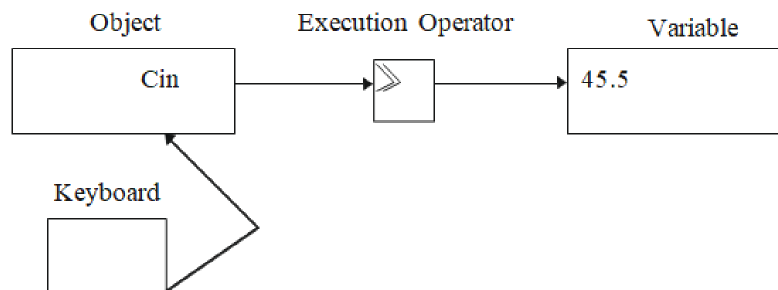


Figure 2.3 Header File

2.13. Output Statement (COUT)

We have used the insertion operator << repeatedly in the last two statements for printing results.

The statement

Cout << “Sum = “ << sum << “\n”;

First sends the string “Sum = “ to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of << n one statement is called cascading. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

Cout << “Sum = “ << sum << “\n”

<< “Average = “ << average << “\n”;

This is one statement but provides two line of output. If you want only one line of output, the statement will be:

Cout << “Sum = “ << sum << “,”

<< “Average = “ << average << “\n”; The output will be:

Sum = 14, average = 7

cout <<, is used to print anything on screen, same as printf in C++ language. cin and cout are same as scanf and printf, only difference is that you do not need to mention format specifiers like, %dfor int etc, in cout & cin.

2.14. Input Statement (CIN)

We can also cascade input iperator >> as shown below:

Cin >> number1 >> number2;

The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to munber1 and 20 to number 2.

2.15. Text Editor

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

Name and version of text editor can vary on different operating systems. Foexample, Notepad will be used on Windows and vim or vi can be used on windows as well as Linux, or UNIX.

The files you create with your editor are called source files and for C++ they typically are named with the extension .cpp, .cp, or .c.

A text editor should be in place to start your C++ programming.

2.16. C++ Compiler

This is an actual C++ compiler, which will be used to compile your source code into final executable program. Most C++ compilers don't care what extension you give to your source code, but if you don't specify otherwise, many will use .cpp by default.

Most frequently used and free available compiler is GNU C/C++ compiler, otherwise you can have compilers either from HP or Solaris if you have the respective Operating Systems.

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 abd 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 basic 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 program using OOPs programming and understanding of writing programs with Object Oriented Programming with C++.

We have discussed the concept of function in C++, its declaration and definition. we have also discussed the concept of class, its declaration and definition. It also explained the ways for creating objects, accessing the data members of the class. We have seen the way to pass objects as arguments to the functions with call by value and call by reference.

2.18. Self Assessment Questions:

Q. 1. what is a function ? How will you define a function in C++ ?

Q. 2. How are the argument data types specified for a C++ function? Explain with Suitable example.

Q. 3. What types of functions are available in C++ ? Explain.

Q. 4. What is recursion? While writing any recursive function what thing(s) must be taken care of ?

Q.5. What is inline function? When will you make a function inline and why ?

Q.6. What is a class? How objects of a class are created ?

Q.7. What is the significance of scope resolution operator (::) ?

Q.8. Define data members, member function, private and public members with example.

Q.9. Define a class student with the following specifications: Adm_no integer

- Sname 20 characters
- Eng, math, science float (marks in three subjects) Total float
- Ctotal() a function to calculate eng + math + science marks

Public member functions of class student

- Takedata() function to accept values for adm_no , sname, marks in eng, math, science and invoke ctotal() to calculate total.
- Showdata() function to display all the data members on the screen.

Q.10. **Define a string data type with the following functionality:**

- A constructor having no parameters,
- Constructors which initialize strings as follows:
- A constructor that creates a string of specific size
- Constructor that initializes using a pointer string
- A copy constructor
- Define the destructor for the class
- It has overloaded operators. (This part of question will be taken up in the later units).

- There is operation for finding length of the string.

Chapter 3
Expression

Objective:

At the end of this session students shall be learning:

- 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 C++ Tokens

3.2. Understand different Data types in C++

3.3. Variables

3.4. C++ Identifiers

3.5. C++ Keywords

3.6. Trigraphs

3.7. Whitespace in C++

3.8. Understand Statement

3.9. Summary

3.10. Self Assessment Questions

3.1 Introduction C++ Tokens

As a programing language, when we consider a C++ program, it can be defined as a collection of objects that communicate via invoking each other's methods. In this chapter we are now considering the use of Object Oriented Concepts of C++, and precisely we are now briefly look into what a class, object, methods, and instant variables mean.

Object − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, eating. An object is an instance of a class.

Class − A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.

Methods − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

Instance Variables − each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

3.1.1 C++ Program Structure

Let us look at a simple code in C++ that would print 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;
}”
```

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 main( )function and causes it to return the value 0 to the calling process.

3.1.2 Compile and Execute C++ Program

In the earlier chapter, we have learned how to write and save the file, compile and run the program. Please follow the steps given below −

- Open a text editor and add the code as above.
- Save the file as: hello.cpp
- Open a command prompt and go to the directory where you saved the file.
- Type program and press enter to compile your code. If there are no errors in your code the command prompt will take you to the next line and would generate an executable file.
- Now, type and execute by running the executable program.
- You will be able to see ' Hello World ' printed on the window.

3.2 Understand different Data types in C++

Let us understand the different Datatypes in C++ (being a programming language). They are used to define type of variables and contents used. Data types define the way you use storage in the programs you write. Data types can be of two types:

1. Built-in Data types
2. User-defined or Abstract Data types

3.2.1 Built-in Data Types

These are the data types which are predefined and are wired directly into the compiler.
For eg: int, char etc.

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)

Example:

char a = 'A'; //

character type int a = 1; //

integer type

float a = 3.14159; // floating point type

double a = 6e-4; // double type (e is for exponential)

3.2.2 User defined

a. Abstract data types

These are the type, that user creates as a class or a structure. In C++ these are classes where as in C language user-defined datatypes were implemented as structures. Other Built in Datatypes in C++

bool Boolean (True or False)

void Without any Value wchar_t

Wide Character

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 Variables?

Variable are used in C++, where we need storage for any value, which will change in program. Variable can be declared in multiple ways each with different memory requirements and functioning. Variable is the name of memory location allocated by the compiler depending upon the data type of the variable (Figure 3.1).

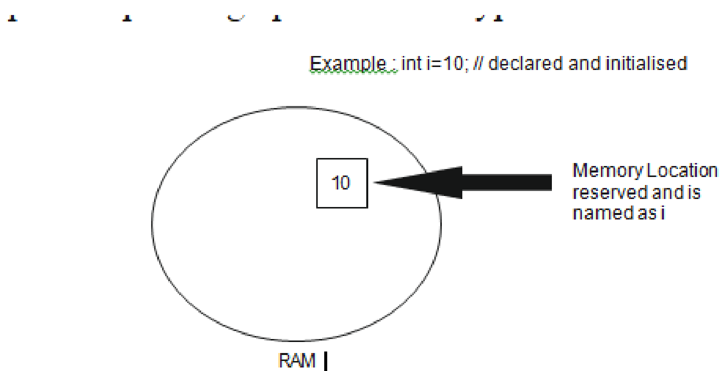


Figure 3.1 Variable

3.3.1 Basic types of Variables

Each variable while declaration must be given a datatype, on which the memory assigned to the variable depends. Following are the basic 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	

Declaration and Initialization

Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

For example:

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;j=20;

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

3.3.2 Scope of Variables

All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable. For most of the cases its between the curly braces,in which variable is declared that a variable exists, not outside it. We will study the storage classes later, but as of now, we can broadly divide variables into two main types,

- **Global Variables**
- **Local variables**

Global variables

Global variables are those, which ar once declared and can be used throughout the lifetime of the program by any class or any function. They must be declared outside the main() function. If only declared, they can be assigned different values at different time in program lifetime. But even if they are declared and initialized at the same time outside the main() function, then also they can be assigned any value 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;

 x=20;  // Initialized again
cout <<"Initialized again with value = "<< x;
}
```

Local Variables

Local variables are the variables which exist only between the curly braces, in which its declared. Outside that they are unavailable and leads to compile time error.

Example :

```
include <iostream>

using namespace std;

int main()
{
int i=10;
if(i<20)            // if condition scope starts

{
int n=100;   // Local variable declared and initialized
}           // if condition scope ends
```

```
cout << n;                    // Compile time error, n not available here
}
```

3.4 C++ Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. 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 allow punctuation characters such as @, $, and % within identifiers. C++ is a case-sensitive programming language. Thus, Manpower and manpower are two different identifiers in C++.

Here are some examples of acceptable identifiers −

```
mohd       zara    abc   move_name  a_123
myname50   _temp   j     a23b9      retVal
```

3.5 C++ Keywords

The following list shows the reserved words in C++. These reserved words may not be used as constant or variable or any other identifier names (Figure 3.2).

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	

Figure 3.2 C++ Keywords

A few characters have an alternative representation, called a trigraph sequence. A trigraph is a three-character sequence that represents a single character and the sequence always starts

with two question marks.

Trigraphs are expanded anywhere they appear, including within string literals and character literals, in comments, and in preprocessor directives.

Following are most frequently used trigraph sequences –

All the compilers do not support trigraphs and they are not advised to be used because of their confusing nature.

3.6 Whitespace in C++

A line containing only whitespace, possibly with a comment, is known as a blank line, and C++ compiler totally ignores it (Figure 3.3).

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

Figure 3.3 Whitespaces in C++

Whitespace is the term used in C++ to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins.

Statement 1

```
int age;
```

In the above statement there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them.

Statement 2

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

In the above statement 2, no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

3.7 Understand Statement

Semicolons and Blocks in C++

While writing the program in C++, the semicolon is used as a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity. For example, following are three different statements −

```
x = y;

y = y + 1;

add(x, y);
```

A block is a set of logically connected statements that are surrounded by opening and closing braces. For example −

```
{

 cout << "Hello World"; // prints Hello World

 return 0;

 }
```

In C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where you put a statement in a line. For example −

```
x = y;

y = y + 1;

add(x, y);
```

is the same as

```
x = y; y = y + 1; add(x, y);
```

3.8 Summary

In this lesson, we discussed the concept and type of constructor and destructor. All the operators that can be overloaded. Even after writing operator overloaded functions, the precedence of operators remains unchanged. The '++' & '--' operators can be used as Postfix or Prefix operators. So, separate functions overloading them for both the different applications

have been shown. we are of a view that Private data of a class can be accessed only in member functions of that class.

3.9 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”.

Chapter 4
Function in C++

Objective:

At the end of this session students shall be learning:

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

Structure:

4.1. Introduction to Function,

4.2. Function definition and declaration

4.3. Arguments to a Function,

4.4. Calling Functions,

4.5. Inline Function,

4.6. Scope rules of function and variable

4.7. Summary

4.8. Self Assessment Questions

4.1. Introduction to the concept of Functions.

Functions are the building blocks of C++ programs where all the program activity occurs. Function is a collection of declarations and statements.

4.1.1. Need for a Function

Monolithic program (a large single list of instructions) becomes difficult to understand. For this reason functions are used. A function has a clearly defined objective (purpose) and a clearly defined interface with other functions in the program. Reduction in program size is another reason for using functions. The functions code is stored in only one place in memory, even though it may be executed as many times as a user needs.

The following program illustrates the use of a function:

```
//to display general message using function #include<iostream.h>
include<conio.h> void main()
{
void disp(); //function prototype clrscr(); //clears the screen disp(); //function call
getch(); //freeze the monitor
}
//function definition void disp()
{
cout<<"Welcome to the GJU of S&T\n"; cout<<"Programming is nothing but logic implementation";
}
```

In this Unit, we will also discuss Class, as important Data Structure of C++. A Class is the backbone of Object-Oriented Computing. It is an abstract data type. We can declare and define data as well as functions in a class. An object is a replica of the class to the exception that it has its own name. A class is a data type and an object is a variable of that type. Classes and objects are the most important features of C++. The class implements OOP features and ties them together.

4.2 FUNCTION DEFINITION AND DECLARATION

In C++, a function must be defined prior to it's use in the program. The function definition contains the code for the function. The function definition for display_message () in program is given below the main () function. The general syntax of a function definition in C++ is shown below.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. ARGUMENTS TO A FUNCTION

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

4.3.1 PASSING ARGUMENTS TO A 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 (Figure 4.1) 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;
```

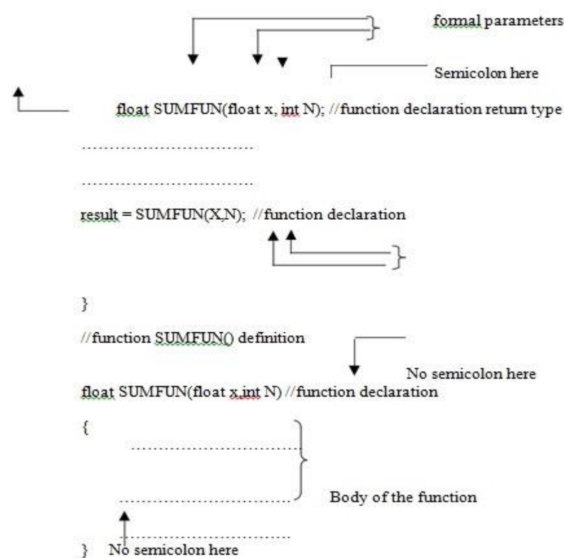


Figure 4.1 Passing Arguments to a function

4.3.2 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. For example.

```
// demonstrate default arguments function

 #include<iostream.h>
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;
}
Void main ()
{
Pattern('');
Pattern ('#',4)'
Pattern (;@;,3);
}
```

4.3.3 CONSTANT ARGUMENTS

A C++ function may have constant arguments(s). These arguments(s) is/are treated as constant(s). These values cannot be modified by the function.

For making the arguments(s) constant to a function, we should use the keyword const as given below in the function prototype :

Void max(const float x, const float y, const float z);

Here, the qualifier const informs the compiler that the arguments(s) having const should not be modified by the function max (). These are quite useful when call by reference method is used for passing arguments.

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 the values of the actual parameters (appearing in the function call) are copied into the formal parameters (appearing in the function definition), i.e., the function creates its own copy of argument values and operates on them. The following program illustrates this concept :

```
//calculation of compound interest using a function #include<iostream.h>
#include<conio.h>   #include<math.h> //for pow()function Void main()
{
Float principal, rate, time; //local variables
Void calculate (float, float, float); //function prototype clrscr(); Cout<<"\nEnter the following
values:\n"; Cout<<"\nPrincipal:";
Cin>>principal; Cout<<"\nRate of interest:"; Cin>>rate;
Cout<<"\nTime period (in yeaers) :"; Cin>>time; Calculate (principal, rate, time); //function
call Getch ();
}
//function definition calculate() Void calculate (float p, float r, float t)
{
Float interest; //local variable Interest = p (pow((1+r/100.0),t))-p; Cout<<"\nCompound
interest is : "<<interest; }
```

4.4.2. Call by Reference: -

A reference provides an alias – an alternate name – for the variable, i.e., the same variable's value can be used by two different names : the original name and the alias name.

In call by reference method, a reference to the actual arguments(s) in the calling program is passed (only variables). So the called function does not create its own copy of original value(s) but works with the original value(s) with different name. Any change in the original data in the called function gets reflected back to the calling function.

It is useful when you want to change the original variables in the calling function by the called function.

```
//Swapping of two numbers using function call by reference #include<iostream.h>
#include<conio.h> void main()
{
clrscr();
int num1,num2;
void swap (int &, int &); //function prototype cin>>num1>>num2; cout<<"\nBefore swapping:\nNum1: "<<num1; cout<<endl<<"num2: "<<num2;
swap(num1,num2); //function call cout<<"\n\nAfter swapping : \Num1: "<<num1; cout<<endl<<"num2: "<<num2; getch();
}
//function fefinition swap() void swap (int & a, int & b)
{
 int temp=a; a=b; b=temp;
}
```

4.5 INLINE FUNCTIONS:

These are the functions designed to speed up program execution. An inline function is expanded (i.e. the function code is replaced when a call to the inline function is made) in the line where it is invoked. You are familiar with the fact that in case of normal functions, the compiler have to jump to another location for the execution of the function and then the control is returned back to the instruction immediately after the function call statement. So execution time taken is more in case of normal functions. There is a memory penalty in the case of an inline function.

The system of inline function is as follows :

```
 inline function_header
{
body of the function
}
```

For example,

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

```
int num1, num2;
cout<<"\Enter the two intergers\n"; cin>>num1>>num2;
min (num1,num2; //function code inserted here

------------

 ------------
}
```

An inline function must be defined before it's invoked, as demonstrated in the example above. In this case, the `min()` function, being inline, won't be directly called during execution; instead, its code will be inserted into `main()` and then compiled.

However, if the inline function's size is substantial, it can incur a significant memory penalty, diminishing its usefulness. In such scenarios, employing a normal function is more practical.

Inlining is ineffective under the following conditions:

1. Functions returning values and containing a loop, switch, or goto statement.
2. Functions lacking return values but containing a return statement.
3. Functions incorporating static variables.
4. Recursive inline functions, where a function is defined in terms of itself.

Advantages of Inline Functions:

1. Minimized Call Overhead: Removes the overhead linked to function calls, such as stack manipulation.
2. Boosted Performance: Enhances execution speed for small, frequently invoked functions.
3. Direct Code Optimization: Enables the compiler to optimize the inline code directly.
4. Maintained Readability: Preserves code clarity while avoiding repetitive code blocks.
5. Encourages Modular Design: Promotes the use of small, modular functions without the performance costs of frequent calls.

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”

4.6.1. Local Scope:- A block in C++ is enclosed by a pair of curly braces i.e., '{' and '}'. The variables declared within the body of the block are called local variables and can be used only within the block. These come into existence when the control enters the block and get destroyed when the control leaves the closing brace. You should note the variable(s) is/are available to all the 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;
}
```

4.6.2. Function Scope: It pertains to the labels declared in a function i.e., a label can be used inside the function in which it is declared. So we can use the same name labels in different functions.

For example,

```
“//function definition add1() void add1(int x,int y,int z)
{
int sum = 0; sum = x+y+z; cout<<sum;
}
//function definition add2() coid add2(float x,float y,float z)
{
Float sum = 0.0; sum = x+y+z; cout<<sum;
}”
```

Here the labels x, y, z and sum in two different functions add1 ( ) and add2 ( ) are declared and used locally.

4.6.3. File Scope: If the declaration of an identifier appears outside all functions, it is available to all the functions in the program and its scope becomes file scope. For Example,

```
int x;
void square (int n)
{
cout<<nn;
}
void main ()
{
int num; cout<<x<<endl; cin>>num;
square(num);
…………..........
}
```

Let us the declarations of variable x and function square ( ) are outside all the functions so these can be accessed from any place inside the program. Such variables/functions are called global.

4.6.4. Class Scope: In C++, every class maintains its won associated scope. The class members are said to have local scope within the class. If the name of a variable is reused by a class member, which already has a file scope, then the variable will be hidden inside the class. Member functions also have class scope.

Inline Functions:- An inline function is expanded in the line where it is invoked.

Member Function:- Private means that they can be accessed only by the functions within the class.

Classes:- When you create the definition of a class you are defining the attributes and behavior of a new type.

Objects:- Declaring a variable of a class type creates an object. You can have many variables of the same type (class).

4.7. Summary

In this Chapter, we have discussed the concept of function in C++, its declaration and definition.

we have also discussed the concept of class, its declaration and definition. It so explained the ways for creating objects, accessing the data members of the class. We have seen the way to pass objects as arguments to the functions with call by value and call by reference.

4.8 Self Assessment Questions

1 What are the advantages of using functions in programming?
2 How do you pass parameters by reference in C++ and what are its benefits?
3 Can you explain with an example how default arguments work in functions?
4 What is the purpose of constant arguments and when should they be used?
5 How does function overloading improve code readability and usability?
6 When should you use inline functions and what are the potential downsides?
7 Can you describe a scenario where using a recursive function is more beneficial than an iterative approach?

Chapter 5
Classes and Objects

Objective:

At the end of this session students shall be learning:

- 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 definition

5.3. Declaration of Objects,

5.4. Friend Classes ,

5.5. Drawbacks of Preprocessors

5.6. Inline function is C++,

5.7. Important point of Inline function,

5.8. Getter and Setter function in C++,

5.9. Limitation of Inline Functions,

5.10. Forwards References in C++

5.11. Function overloading in C++

5.12. Summary

5.13 Self Assessment Questions

5.1. Introduction of a Class

A class in C++ combines related data and functions together. It makes a data type which is used for creating objects of this type.

Classesrepresent real world entities that have bothdata type properties (characteristics) and associated operations (behavior).

The syntax of a class definition is shown below:

Class name_of _class

“{

: variable declaration; // data member

Function declaration; // Member Function (Method) protected: Variable declaration;

Function declaration;

public : variable declaration;

{

Function declaration;

};”

Here, the keyword class specifies that we are using a new data type and is followed by the class name.

The body of the class has two keywords namely :

(i) “private” (ii) “public”

In C++, the keywords private and public are called access specifiers. The data hiding concept in C++ is achieved by using the keyword private. Private data and functions can only be accessed from within the class itself. Public data and functions are accessible outside the class also. This is shown below (Figure 5.1):

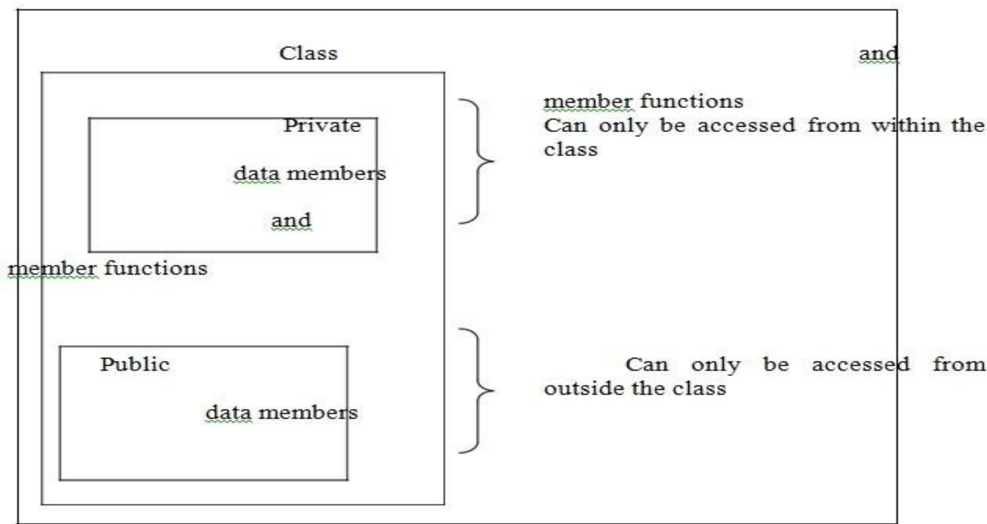


Figure 5.1 Class

Data hiding not mean the security technique used for protecting computer databases. The security measure is used to protect unauthorized users from performing any operation (read/write or modify) on the data.

The data declared under **Private** section are hidden and safe from accidental manipulation. Though the user can use the private data but not by accident.

The functions that operate on the data are generally **public** so that they can be accessed from outside the class but this is not a rule that we must follow.

5.2. Member Function Definition

The specification of a class is divided into two parts:

1. Class Definition: This includes the declaration of data members and member functions.
2. Class Method Definitions: This involves writing the actual code for the member functions of the class.

Previously, we covered the syntax for defining a class along with an example. In C++, member functions can be implemented in two distinct ways:

- Within the Class Definition: Coding the member functions directly inside the class declaration.
- Outside the Class Definition: Implementing the member functions outside the class declaration using the scope resolution operator (::).

The code of the function is same in both the cases, but the function header is different as explained below:

5.2.1. Inside Class Definition:

When a member function is defined inside a class, we do not require to place a membership label along with the function name. We use only small functions inside the class definition and such functions are known as inline functions.

In case of inline function the compiler inserts the code of the body of the function at the place where it is invoked (called) and in doing so the program execution is faster but memory penalty is there.

5.2.2. Outside Class Definition Using Scope Resolution Operator (::) :

In this case the function's full name (qualified_name) is written as shown:

Name_of_the_class :: function_name

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

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

Here the operator::known as scope resolution operator helps in defining the member function outside the class. Earlier the scope resolution operator(::)was ised om situations where a global variable exists with the same name as a local variable and it identifies the global variable.

5.3. Declaration Of Objects As Instances Of A Class

The objects of a class are declared after the class definition. One must remember that a class definition does not define any objects of its type, but it defines the properties of a class. For utilizing the defined class, we need variables of the class type. For example,

Largest ob1,ob2; //object declaration will create two objects ob1 and ob2 of largest class type.

As mentioned earlier, in C++ the variables of a class are known as objects. These are declared like a simple variable i.e., like fundamental data types.

In C++, all the member functions of a class are created and stored when the class is defined and this memory space can be accessed by all the objects related to that class.

Memory space is allocated separately to each object for their data members. Member variables store different values for different objects of a class.

5.3.1. Accessing Members From Object(S)

After defining a class and creating a class variable i.e., object we can access the data members and member functions of the class. Because the data members and member functions are parts of the class, we must access these using the variables we created. For functions are parts of the class, we must access these using the variable 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()
}
};”
student ob; //class variable (object) created
- - - - -
- - - - -

Ob.init_data(); //Access the member function ob.display_data(); //Access the member function - - - - -
- - - - -
```

Here, the data members can be accessed in the member functions as these 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

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

static data members and static member function in a class.

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

(i) Only a single copy of the static data member is used by all the objects.

(ii) It can be used within the class but its lifetime is the whole program. For making a data member static, we require :

(a) Declare it within the class.

(b) Define it outside the class. For example

```
Class student
"{
Static int count; //declaration within class
- - - - - - - - - - -
   - - - - - - - - - - -
   - - - - - - - - - - -
};"
```

The static data member is defined outside the class as :

"int student :: count;" //definition outside class The definition outside the class is a must.

We can also initialize the static data member at the time of its definition as: "int student :: count = 0;"

If we define three objects as : student obj1, obj2, obj3;

Static Member Function: A static member function can access only the static members of a class. We can do so by putting the keyword static before the name of the function while declaring it for example,

```
Class student
"{
Static int count;
-----------------
public :

- - - - - - - - - - -
- - - - - - - - - - -

static void showcount (void) //static member function
{
Cout<<"count="<<count<<"\n";
```

}
int student ::count=0;
};"
Here we have put the keyword static before the name of the function shwocount ().
In C++, a static member function fifers from the other member functions in the following ways:
(i) Only "static members (functions or variables)" of the same class can be accessed by a static member function.
(ii) It is called by using the name of the class rather than an object as given below:

Name_of_the_class :: function_name For example,
student::showcount();

5.4. Friend Classes

In C++ , a class can be made a friend to another class. For example, class TWO; // forward declaration of the class TWO
class ONE
"{
………………………
…………….
public:
……………..
……………..
friend class TWO; // class TWO declared as friend of class ONE
};"
Now from class TWO , all the member of class ONE can be accessed.
Friend functions are actually not class member function. Friend functions are made to give privateaccess to non-class functions. You can declare a global function as friend, or a member function of other class as friend.
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
}"
```

Hence, friend functions can access private data members by creating object of the class. Similarly we can also make function of some other class as friend, or we can also make an entire class as 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 we make a class as friend, all its member functions automatically become friend functions. Friend Functions is a reason, why C++ is not called as a pure Object Oriented language. As such, it violates the concept of Encapsulation.

All the member functions defined inside the class definition are by default declared as Inline. Let us have some background knowledge about these functions.

You must remember Preprocessors from C language. Inline functions in C++ do the same thing what Macros did in C language. Preprocessors/Macros were not used in C++ because they had some drawbacks.

5.5. Drawbacks of Preprocessors/Macros in C++

In Macro, we define certain variable with its value at the beginning of the program, and everywhere inside the program where we use that variable, its replaced by its value on compilation.

1) Problem with spacing:

Let us try to understand this problem using an example,

#define G (y) (y+1)

Here we have defined a Macro with name G(y), which is to be replaced by its value, that is (y+1)during compilation. But, what actually happens when we call G(y),

G(1) // Macro will replace it

the preprocessor will expand it like,

(y) (y+1) (1)

You must be thinking why this happened, this happened because of the spacing in Macro definition. Hence big functions with several expressions can never be used with macro, so Inline functions were introduced in C++.

2) Complex Argument Problem

In some cases such Macro expressions work fine for certain arguments but when we use complex arguments problems start arising.

#define MAX(x,y) x>y?1:0 Now if we use the expression,

if(MAX(a&0x0f, 0x0f)) // Complex Argument Macro will Expand to, if(a&0x0f > 0x0f ? 1:0)

Here precedence of operators will lead to problem, because precedence of & is lower than that of >, so the macro evaluation will surprise you. This problem can be solved though using parenthesis, but still for bigger expressions problems will arise.

3) No way to access Private Members of Class

With Macros, in C++ you can never access private variables, so you will have to make those members public, which will expose the implementation.

class Y

{

```
int x; public :
#define VAL(Y::x)   // Its an Error
}
```

5.6. Inline Functions in C++

Inline functions are actual functions, which are copied everywhere during compilation, like preprocessor macro, so the overhead of function calling is reduced. All the functions defined inside class definition are by default inline, but you can also make any non-class function inline by using keyword inline with them.

For an inline function, declaration and definition must be done together. For example, inline void fun(int a)

```
{
return a++;
}
```

5.7. Some Important points about Inline Functions:

1. We must keep inline functions small, small inline functions have better efficiency.
2. Inline functions do increase efficiency, but we should not make all the functions inline. Because if we make large functions inline, it may lead to code bloat, and might affect the speed too.
3. Hence, it is adviced to define large functions outside the class definition using scope resolution ::operator, because if we define such functions inside class definition, then they become inline automatically.
4. Inline functions are kept in the Symbol Table by the compiler, and all the call for such functions is taken care at compile time.

5.8. Getter and Setter Functions in C++

We have already studied this in the topic accessing private data variables inside a class. We use acce class 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.
```

Here getPrice() and setPrice() are inline functions, and are made to access the private data members of the class Auto. The function getPrice(), in this case is called Getter or Accessor function and the function setPrice() is a Setter or Mutator function. There can be overlaoded Accessor and Mutator functions too. We will study overloading functions in next topic.

5.9. Limitations of Inline Functions:

1. Large Inline functions cause Cache misses and affect performance negatively.
2. **Compilation Overhead**: Copying the function body everywhere during compilation introduces overhead. While this is negligible for small programs, it can significantly impact performance in larger codebases.
3. **Function Address Requirement**: When the address of a function is needed in a program, the compiler cannot inline such functions. This is because inlining does not allocate storage for the function, instead keeping it in the symbol table, which precludes the assignment of a physical address.

5.10. Understanding Forward References in C++

All the inline functions are evaluated by the compiler, at the end of class declaration. 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 must be thinking that this will lead to compile time error, but in this case it will work, because no inline function in a class is evaluated until the closing braces of class declaration.

5.11. Function Overloading in C++

If any class have multiple functions with same names but different parameters then they are said to be overloaded. Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.

Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments, then you can simply overload the function.

5.11.1. Different ways to Overload a Function

1. By changing number of Arguments.
2. By having different types of argument.

5.11.2. Function Overloading: Different Number of Arguments

In this type of function overloading we define two functions with same names but different number of parameters of the same type. For example, in the below mentioned program we have made two sum() functions to return 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 sum() function is said to overloaded, as it has two definition, 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
```

5.11.3. Function Overloading: Different Datatype of Arguments

In this type of overloading we define two or more functions with same name and same number of parameters, but the type of parameter is different. For example in this program, we have two sum() function, first one gets two integer arguments and second one gets two double arguments.

```
// first definition int sum(int x, int y)
{
cout<< x+y;
}
// second overloaded defintion double sum(double x, double y)
{
cout << x+y;
}
int main()
{
sum (10,20); sum(10.5,20.5);
} 30
31.0
```

When we mention a default value for a parameter while declaring the function, it is said to be as default argument. In this case, even if we make a call to the function without passing any value for that parameter, the function will take the default value specified.

```
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 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.

5.11.5. Rules for using Default Arguments

1. Only the last argument must be given default value. You cannot have a default argument followed by 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 give any value a default value to argument, compatible with its datatype.

5.11.6. 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.

In this chapter, you have been exposed to the concepts of base class and derived classes. A derived class is a class which includes the member of another class. This concept is also known as inheritance. When a derived class has more than one direct base class, then it is called

Multiple Inheritance. There were three types of inheritance. We can also declare classes as members of another class. We have also touched on the concept 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 is a Classes and Objects? Explain the procedure for the same.
4. What is a Constructors? What is a Destructor? Illustrate with example Constructor and destructions.
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++:
    - i. Simple functions
    - ii. Static functions
    - iii. Const functions
    - iv. Inline functions
    - v. Friend functions
8. Drawbacks of Preprocessors/Macros in C++.
9. what is 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. Illustrate with Different ways to Overload a Function
    - i. By changing number of Arguments.
    - ii. By having different types of argument.
13. Explain the types of Constructors in C++
    - i. Default Constructor
    - ii. Parameterized Constructor
    - iii. Copy Constructor

Chapter 6

Constructor and Destructor

Objective:

At the end of this session students shall be learning:

- 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

6.2. Declaration and definition Constructor

6.3. Type of Constructor

6.4. Constructors and Primitive types

6.5. Declaration and definition destructor

6.6. Special Characteristics of

6.7. Summary

6.8. Self Assessment Questions

6.1 Introduction

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 enhancement of 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 (lengthbreadth);
}
};
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 performing the role of member data initialization, constructors are no different from other functions. This included overloading also. In fact, it is very common to find overloaded constructors. For example, 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=side1side2; else
ar=6.14radiusradius;
cout<<"\n\nArea of the "<<shape<<"is :"<<ar<<"sq.units\n";
}
};
Void main()
{
Clrscr();
Figure circle(10.0); //objrct initialized using constructor Figure

rectangle(15.0,20.6);//objrct initialized using onstructor
Figure Triangle(6.0, 4.0, 5.0); //objrct initialized using constructor Rectangle.area();
Triangle.area(); Getch();//freeze the monitror"
```

6.3.2 Copy Constructor

It is of the form classname (classname &) and used for the 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 Dynamic Initialization of Objects

In C++, the class objects can be initialized at run time (dynamically). We have the flexibility of providing initial values at execution time. The following program illustrates this concept:

```
//Illustration of dynamic initialization of objects #include <iostream.h>
#include <conio.h> Class employee
{
Int empl_no; Float salary; Public:
Employee() //default constructor
{}
Employee(int empno,float s)//constructor with arguments { Empl_no=empno;
Salary=s;
}
Employee (employee &emp)//copy constructor
{
Cout<<"\ncopy constructor working\n"; Empl_no=emp.empl_no; Salary=emp.salary;
}
Void display (void)
{
Cout<<"\nEmp.No:"<<empl_no<<"salary:"<<salary<<end1;
}
};
Void main()
{
int eno; float sal; clrscr();
cout<<"Enter the employee number and salary\n"; cin>>eno>>sal;
```

```
employee obj1(eno,sal);//dynamic initialization of object cout<<”\nEnter the employee
number and salary\n”; cin>eno>>sal; employee obj2(eno,sal); //dynamic initialization of
object obj1.display(); //function called
employee obj3=obj2; //copy constructor called obj6.display();
getch();
}
```

6.4. 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.

For 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
```

6.6.5 Constructor with Default Arguments

In C++, we can define constructor s with default arguments. For example, The following code segment shows a constructor with default arguments: 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.

6.4 Special Characteristics Of Constructors

These have some special characteristics. These are given below:

(i) These are called automatically when the objects are created.

(ii) All objects of the class having a constructor are initialized before some use.

(iii) These should be declared in the public section for availability to all the functions.

(iv) Return type (not even void) cannot be specified for constructors.

(v) These cannot be inherited, but a derived class can call the base class constructor.

(vi) These cannot be static.

(vii) Default and copy constructors are generated by the compiler wherever required. Generated constructors are public.

(viii) These can have default arguments as other C++ functions.

(ix) A constructor can call member functions of its class.

(x) An object of a class with a constructor cannot be used as a member of a union.

(xi) A constructor can call member functions of its class.

(xii) We can use a constructor to create new objects of its class type by using the syntax. Name_of_the_class (expresson_list) For example,

Employee obj3 = obj2; // see program 10.5

Or even

Employee obj3 = employee (1002, 35000); //explicit call

(xiii) The make implicit calls to the memory allocation and deallocation operators new and delete.

(xiv) These cannot be virtual.

6.5 Declaration and Definition of a Destructor

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();
}"
```

6.6 Special Characteristics of Destructors

Some of the characteristics associated with destructors are :

1. Automatic Execution: Destructors are invoked automatically when an object exits scope or is explicitly deallocated.
2. Parameterless: Destructors cannot accept any parameters.
3. No Return Type: Destructors do not return any value.
4. Singular per Class: Each class is limited to a single destructor.
5. Non-Inheritable: Destructors do not get inherited in the conventional sense.
6. Resource Cleanup: Destructors handle the release of resources allocated to an object during its lifecycle.
7. Destruction Order: In inheritance hierarchies, destructors are executed in the reverse sequence of constructors.
8. Not Explicitly Callable: Destructors are not intended to be called directly by the user.
9. Exception Handling: Destructors are executed during stack unwinding when an exception is thrown, ensuring resource release.
10. Chained Destruction: In derived classes, destructors invoke base class destructors in sequence.
11. Consistency and Stability: Destructors help maintain resource management consistency and prevent resource leaks, ensuring program stability.

6.7. Summary

In this chapter, we explored the concepts of constructors and destructors, detailing their various types and roles within object-oriented programming. Additionally, we examined the different operators that are eligible for overloading. It's crucial to note that even after defining custom operator overloads, the intrinsic precedence rules of these operators remain unaffected. We discussed how the increment (++) and decrement (--) operators can function in both postfix and prefix forms. As a result, we demonstrated the necessity of defining separate overload functions for each usage. Furthermore, we reiterated the principle that a class's private data members can only be accessed through its member functions, ensuring encapsulation and data integrity.

6.8. Self Assessment Questions

1. Explain the types of Constructors in C++
    - Default Constructor, Parameterized Constructor, Copy Constructor
2. Example of how to make use Constructor and Destructor.
3. Define and illustrate the Static Keyword in C++:
    - Static variable in functions, Static Class Objects
    - Static member Variable in class, Static Methods in class
4. Difference between Reference and Pointer in C++.
5. Explain with example Copy Constructor in C++ :
    - Shallow
    - Deep

Chapter 7
Operator Overloading and Type Conversion

Objective:

At the end of this session students shall be learning:

- 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 to overloading

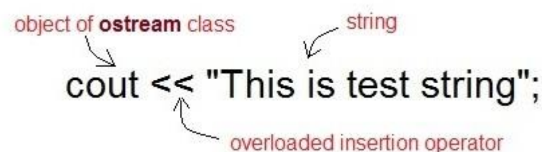
7.2. Types Conversion

7.3. Summary

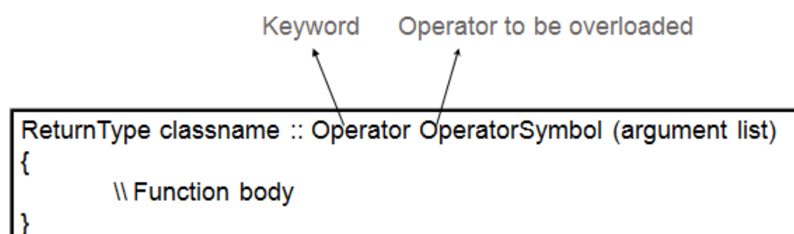
7.4. Self Assessment Questions

7.1. Introduction to Overloading

Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.



7.1.1. Operator Overloading Syntax



Almost any operator can be overloaded in C++. However there are few operator which cannot be overloaded. Operator that are not overloaded are follows:

- "scope operator - ::"
- "size of"
- "member selector - ."
- "member pointer selector - "
- "ternary operator - ?:"

7.1.2. Implementing Operator Overloading in C++

Operator overloading can be done by implementing a function which can be:

1. "Member Function"
2. "Non-Member Function"
3. "Friend Function"

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.

Operator overloading function can be made friend function if it needs access to the private and protected members of class.

7.1.3. Restrictions on Operator Overloading in C++

Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and Associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of a procedure. You cannot change how integers are added.

7.1.4. Operator Overloading Examples in C++

Almost all the operators can be overloaded in infinite different ways. Following are some examples to learn more about operator overloading. All the examples are closely connected.

a. Overloading Arithmetic Operator in C++

Arithmetic operator are most commonly used operator in C++. Almost all arithmetic operator can be overloaded to perform arithmetic operation on user-defined data type. In the below

example we have overridden the + operator, to add to Time(hh:mm:ss) objects.

Example: overloading + Operator to add two Time class object #include <iostream.h>

```
#include <conio.h> class Time
{
int h,m,s; public:
Time()
{
h=0, m=0; s=0;
}
void setTime(); void show()
{
cout<< h<< ":"<< m<< ":"<< s;
}
//overloading '+' operator Time operator+(time);
};

Time Time::operator+(Time t1)          //operator function
{
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()
{
cout << "\n Enter the hour(0-11) "; cin >> h;
cout << "\n Enter the minute(0-59) "; cin >> m;
cout << "\n Enter the second(0-59) "; cin >> s;
}
void main()
{
```

```
Time t1,t2,t3;
cout << "\n Enter the first time "; t1.setTime();
cout << "\n Enter the second time "; t2.setTime();

t3 = t1 + t2;     //adding of two time object using '+' operator cout << "\n First time ";
t1.show();
cout << "\n Second time ";


t2.show();
cout << "\n Sum of times "; t3.show();
getch();
}
```

While normal addition of two numbers return the sumation result. In the case above we have overloaded the + operator, to perform addition of two Time class objects. We add the seconds, minutes and hour values separately to return the new value of time.

In the setTime() funtion we are separately asking the user to enter the values for hour, minute and second, and then we are setting those values to the Time class object.

For inputs, t1 as 01:20:30(1 hour, 20 minute, 30 seconds) and t2 as 02:15:25(2 hour, 15 minute, 25 seconds), the output for the above program will be:

1:20:30

2:15:25

3:35:55

First two are values of t1 and t2 and the third is the result of their addition.

7.1.5. Overloading I/O operator in C++

The first question before learning how to override the I/O operator should be, why we need to override the I/O operators. Following are a few cases, where overloading the I/O operator proves useful:

We can overload output operator << to print values for user defined datatypes. We can overload output operator >> to input values for user defined datatypes.

In case of input/output operator overloading, left operand will be of types ostream& and istream&

Also, when overloading these operators, we must make sure that the functions must be a Non-

Member function because left operand is not an object of the class. And it must be a friend function to access private data members.

You have seen above that << operator is overloaded with ostream class object cout to print primitive datatype values to the screen, which is the default behaviors of << operator, when used with cout. In other words, it is already overloaded by default.

Similarly we can overload << operator in our class to print user-defined datatypes to screen. For example we can overload << in our Time class to display the value of Time object using cout rather than writing a custom member function like show() to print the value of Time class objects.

```
Time t1(3,15,48);
// like this, directly cout << t1;
```

When the operator does not modify its operands, the best way to overload the operator is via friend function.

Example: overloading << Operator to print Class Object We will now overload the << operator in the Time class, #include<iostream.h>

```
#include<conio.h> 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 ostream& operator << (ostream &out, Time &tm);
};
// define the overloaded function
ostream& operator << (ostream &out, Time &tm)
{
out << "Time is: " << tm.hr << " hour : " << tm.min << " min : " << tm.sec << " sec"; return out;
```

```
}
void main()
{
Time tm(3,15,45); cout << tm;
}
```

Time is: 3 hour : 15 min : 45 sec

This is simplied in languages like Core Java, where all you need to do in a class is override the toString() method of the String class, and you can define how to print the object of that class

7.1.6. Overloading Relational Operator in C++

You can also overload relational operators like == , != , >= , <= etc. to compare two object of any class.

Let's take a quick example by overloading the == operator in the Time class to directly compare two objects of Time class.

```
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";
}
}
```

Both the time values are not equal

As the hour value of object t1 is 3 and for object t2 it is 4, hence they are not equal.

7.1.7. Copy Constructor vs. Assignment Operator (=)

Assignment operator is used to copy the values from one object to another already existing object. For example:

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

Whereas, Copy constructor is a special constructor that initializes a new object from an existing object.

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

In case of Copy constructor, we provide the object to be copied as an argument to the constructor. Also, we first need to define a copy constructor in our class. We have covered Copy constructor in detail here: Copy Constructor in C++.

For defining an additional task to an operator, we must mention what is means in relation to the class to which it (the operator) is applied. The operator function helps us in doing so.

The Syntax of declaration of an Operator function is as follows: Operator Operator_name

For example, suppose that we want to declare an Operator function for '='. We can do it as follows:

operator =

A Binary Operator can be defined either a member function taking one argument or a

global function taking one arguments. For a Binary Operator X, a X b can be interpreted as either an operator X (b) or operator X (a, b).

For a Prefix unary operator Y, Ya can be interpreted as either a.operator Y ( ) or Operator Y (a). For a Postfix unary operator Z, aZ can be interpreted as either a.operator Z(int) or Operator (Z(a),int).

The operator functions namely operator=, operator [ ], operator ( ) and operator? must be non-static member functions. Due to this, their first operands will be lvalues.

An operator function should be either a member or take at least one class object argument. The operators new and delete need not follow the rule. Also, an operator function, which needs to accept a basic type as its first argument, cannot be a member function. Some examples of declarations of operator functions are given below:

class P

{

P operator ++ (int);//Postfix increment P operator ++ ( ); //Prefix increment P operator || (P);

//Binary OR

}

Some examples of Global Operator Functions are given below: P operator – (P); // Prefix Unary minus

P operator – (P, P); // Binary "minus" P operator - - (P &, int); // Postfix Decrement

We can declare these Global Operator Functions as being friends of any other class. Examples of operator overloading:

7.1.8. Operator overloading using friend. Class time

{

int r;

int i;

public:

friend time operator + (const time &x, const time &y);

// operator overloading using friend

time ( ) { r = i = 0;} time (int x, int y) {r = x; i = y;}

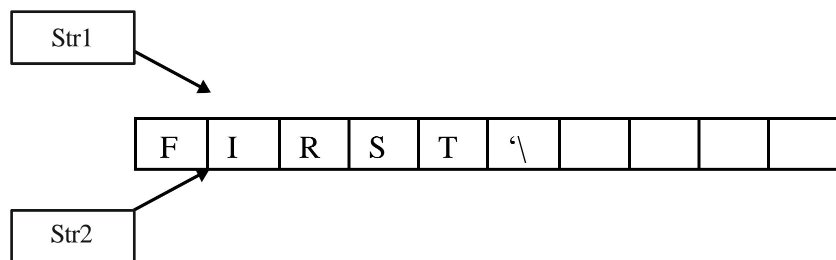
```
};
time operator + (const time &x, const time &y)
{
time z;
z.r = x.r +y.r;
z.i = x.i + y.i; return z;
}
main ( )
{
time x,y,z;
x = time (5,6);
y = time (7,8);
z = time (9, 10);
z = x+y; // addition using friend function +
}
```

7.1.9. Operator overloading using member function:

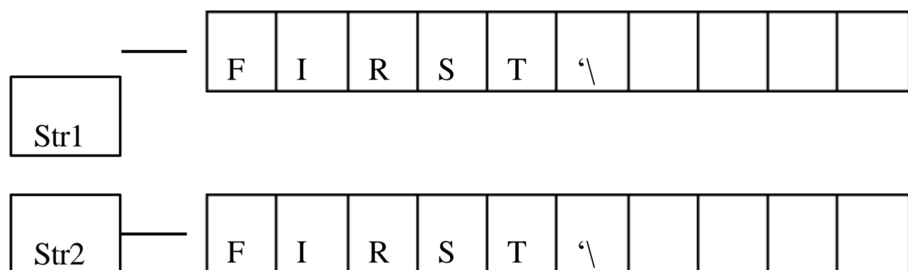
```
Class abc
{
char str;
int len ; // Present length of the string
int max_length; // (maximum space allocated to string) public:
abc ( ); // black string of length 0 of maximum allowed length of size 10. abc (const
abc &s ) ;// copy constructor
~ abc ( ) {delete str;}
int operator = = (const abc &s ) const; // check for equality abc & operator = (const abc &s ); //
overloaded assignment operator
friend abc operator + (const abc &s1, const abc &s2);
} // string concatenation abc:: abc ()
{
max_length = 10;
str = new char [ max_length]; len = 0; str [0] = '\0';
}
```

```
abc :: abc (const abc &s )
{
len = s. len;
max_length = s.max_length; str = new char [max_length];
strcpy (str, s.str); // physical copying in the new location.
}
```

[Note: Please note the need of explicit copy constructor as we are using pointers. For example, if a string object containing string "first" is to be used to initialise a new string and if we do not use copy constructor then will cause:



That is two pointers pointing to one instance of allocated memory, this will create problem if we just want to modify the current value of one of the string only. Even destruction of one string will create problem. That is why we need to create separate space for the pointed string as:



Thus, we have explicitly written the copy constructor. We have also written the explicit destructor for the class.

This will not be a problem if we do not use pointers.

```
abc :: ~ abc ( )
{
delete str;
}
abc & abc :: operator = (const abc &s )
{
```

```
if (this ! = &s) // if the left and right hand variables are different

{

len = s.len;

max_length = s.max-length;

delete str; // get rid of old memory space allocated to this string str = new

char [max_length];   // create new locations

strcpy (str, s.str); // copy the content using string copy function

}

return this;

}
```

//Please note the use of this operator which is a pointer to object that // invokes the call to this assignment operator function.

```
inline int abc :: operator == (const abc &s ) const

{

//uses string comparison function return strcmp (str,s.str);

}


abc                         abc:: operator + (const

abc &s ) abc s3;

s6.len = len + s.len; s6.max_length = s6.len;

char newstr = new char [length + 1]; strcpy (newstr,

s.str);

strcat (newstr,str); s6.str =

newstr; return (s3);

}
```

Overloading << operator:

To overload << operator, the following function may be used: Ostream &

```
operator << (ostream &s, const abc &x )

{s<< "The String is:" <<x; } return s;}
```

You can write appropriate main function and use the above overloaded operators as shown in the complex number example.

7.1.10. Assignment And Initialisation Consider the following class:

```
class student
{
char name; int rollno; public:
student ( ) {name = new char [20];}
~        student ( ) {delete [ ] name;}
};
int f ( )
{ student S1, S2; cin >> S1; cin >> S2; S1 = S2;
}
```

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

Defining assignment of strings as follows can solve this problem, class student

```
{
Public:
char name; int rollno;
student ( ) {name = new char [20];}
~        student ( ) {delete [ ] name ;} student & operator = (const student & )
}
student & student :: Operator = (const student &e)
{
if (this ! =&e) delete [] name;
name = new char [20]; strcpy(name, name);
}
return this;}
```

7.2 Type Conversions

We have overloaded several kinds of operators but we haven't considered the assignment operator (=). It is a very special operator having complex properties. We know that = operator assigns values form one variable to another or assigns the value of 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 used in program 11.2 earlier, 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 by the compiler provided the data type on both sides of = are of 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 tell to the compiler for the solution.

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

(i) Conversion form basic type to class type.

(ii)Conversion from class type to basic type.

(iii) 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 very easy. 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;
}
};
```

The following conversion statements can be coded in a function : distance dist1; //object dist1

created int length = 20; dist1=length; //int to class type

After the execution of above statements, the feet member of dist1 will have a value of 1

and inches member a value of 8, meaning 1 feet and 8 inches.

A class object has been used as the left hand operand of = operator, so the type conversion can also be done by using an overloaded = operator in C++.

(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. The syntax of the conversion function is given below:

```
Operator typename()
{
.......
.........................//statements
}
```

Here, the function converts a class type data to typename. For example, the operator float ( ) converts a class type to type float, the operator int ( ) converts a class type object to type int. For example,

```
matrix :: operator float ()
{
float sum = 0.0; for(int
i=0;i<m;i++)
{
for (int j=0; j<n; j++)
sum=sum+a[i][j]a[i][j];
}
Return sqrt(sum); //norm of the matrix
}
```

Here, the function finds the norm of the matrix (Norm is the square root of the sum of the squares of the matrix elements). We can use the operator float ( ) as given below :

float norm = float (arr); or

float norm = arr;

where arr is an object of type matrix. When a class type to a basic type conversion is required, the compiler will call the casting operator function for performing this task.

The following conditions should be satisfied by the casting operator function :

(a) It must not have any argument

(b) It must be a class member

(c) It must not specify a return type.

(i) One Class Type to Another Class Type

There may be some situations when we want to convert one class type data to another class type data. For example,

Obj2 = obj1; //different type of objects

Suppose obj1 is an object of class studdata and obj2 is that of class result. We are converting the class studdata data type to class result type data and the value is assigned to obj2. Here studdata is known as source class and result is known as the destination class.

The above conversion can be performed in two ways :

(a) Using a constructor.

(b) Using a conversion function.

When we need to convert a class, a casting operator function can be used i.e. source class. The source class performs the conversion and result is given to the object of destination class.

If we take a single-argument constructor function for converting the argument's type to the class type (whose member it is). So the argument is of the source class and being passed to the destination class for the purpose of conversion. Therefore it is compulsory that the conversion constructor be kept in the destination class.

7.3 Summary

In this lesson, we discussed the operators that can be overloaded. Even after writing operator overloaded functions, the precedence of operators remains unchanged. The '++' & '--' operators can be used as Postfix or Prefix operators. So, separate functions overloading them for both the different applications have been shown. We are of a view that Private data of a class can be accessed only in member functions of that class.

7.4 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 constructor function in a class.
8. Design a class having the constructor and destructor functions that shiukd display the number of object being created or destroyed of this class type.
9. Write a C++ program, to find the factorial of a number using a constructor and a destructor (generating the message "you have done it")
10. Define a class "string" with members to initialize and determine the length of the string. Overload the operators '+' and '+=' for the class "string.

Chapter 8
Inheritance

Objective:

At the end of this session students shall be learning:

- 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 in C++
8.3. Multiple Inheritance in C++
8.4. Hierarchical inheritance in C++
8.5. Multilevel Inheritance in C++
8.6. Hybrid Inheritance in C++
8.7. Constructor call in Multiple Inheritance
8.8. Upcasting in C++
8.9. Function that are never Inherited
8.10. Inheritance and static function in C++
8.11. Hybrid Inheritance and Virtual Class
8.12. Hybrid Inheritance and Contructor
8.13. Summary
8.14. Self Assessment Questions

8.1 Introduction OOPs – Inheritance

Inheritance is a way to reuse once written code again and again. The class which is inherited is called the Base class & the class which inherits is called the Derived class. They are also called parent and child class. So when, a derived class inherits a base class, the derived class can use all the functions which are defined in base class, hence making code reusable.

Considering HumanBeing a class, which has properties like hands, legs, eyes etc, and functions like walk, talk, eat, see etc. Male and Female are also classes, but most of the properties and functions are included in HumanBeing, hence they can inherit everything from class HumanBeing using the concept of Inheritance.

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class. (If class A is inherited by class B, then class B is subclass of class A. We will learn about inheritance later.)

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

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the Parent or Base or Super class. And, the class which inherits properties of other class is called Child or Derived or Sub class.

Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

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.

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. The real appeal and power of the inheritance mechanism is that it.

Allows the programmer to reuse a class i.e almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduced any undesirable side-effects into the rest of classes. We have already learned the process of Inheritance while discussing the Object-Oriented Programming in C++.

8.1.1. Basic Syntax of Inheritance

class Subclass_name : access_mode Superclass_name

While defining a subclass like this, the super class 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, privtate or protected.

Example of Inheritance

Whenever we want to use something from an existing class in a new class, we can use the concept on Inheritance. Here is a simple example,

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;
}  4 1
```

Inheritance allows a class to include the members of other classes without repetition of members. There were three ways to 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 = 386;

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.

8.1.2. 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 sub class 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 becomes protected members of Sub class.

class subclass : protected Superclass

Table showing all the Visibility Modes

	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

Inheritance relationship is the highest relationship that can be represented in C++. It is a powerful way of representing a hierarchical relationship directly. The real appeal and power of the inheritance mechanism is that it allows us to reuse a class that is almost, but not exactly, what we want and to tailor the class in a way that is does not introduce any unwanted side effects into the rest of the class. We must review the attributes and operations of the classes and prepare an inheritance relationship:

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

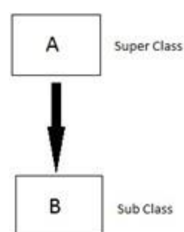
8.1.3. Types of Inheritance in C++

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)

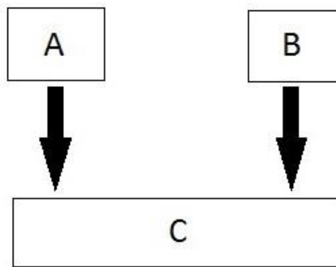
8.2. Single Inheritance in C++

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



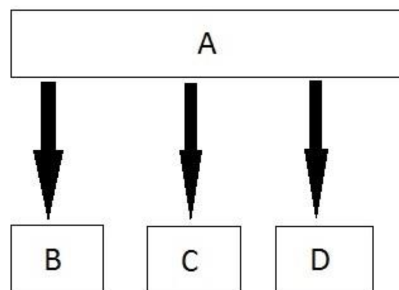
8.3. Multiple Inheritances in C++

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



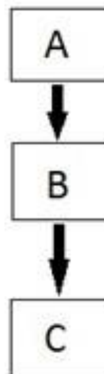
8.4. Hierarchical Inheritance in C++

In this type of inheritance, multiple derived classes inherits from a single base class..



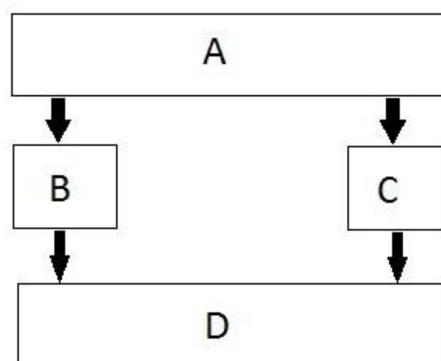
8.5. Multilevel Inheritance in C++

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



8.6. Hybrid (Virtual) Inheritance in C++

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



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 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);
}
```

Base default constructor Base default constructor Derived default constructor Base default constructor

Derived parameterized constructor

You will see in the above example that with both the object creation of the Derived class, Base class's default constructor is called.

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) ;
}
```

Base Parameterized Constructor Derived Parameterized Constructor

8.6.3. Why is Base class Constructor called inside Derived class?

Constructors have a special job of initializing the object properly. A Derived class constructor has access only to its own class members, but Derived class objects also have inherited property of Base class, and only base class constructor can properly initialize base class members. Hence all the constructors are called, else object wouldn't be constructed properly.

8.7. Constructor call in Multiple Inheritance in C++

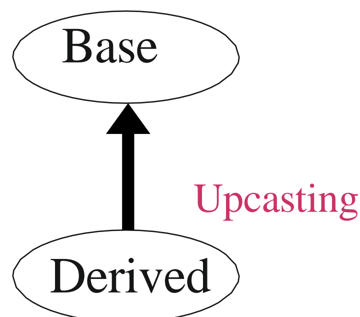
It is almost the same, all the Base class's constructors are called inside derived class's constructor, in the same order in which they are inherited.

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
};
```

Now by adding virtual keyword, we tell compiler to call any one out of the two show() functions.

8.12. Hybrid Inheritance and Constructor call

As we all know that whenever a derived class object is instantiated, the base class constructor is always called. But in case of Hybrid Inheritance, as discussed in above example, if we create an instance of class D, then following constructors will be called:

before class D's constructor, constructors of its super classes will be called, hence constructors of class B, class C and class A will be called.

when constructors of class B and class C are called, they will again make a call to their super class's constructor.

This will result in multiple calls to the constructor of class A, which is undesirable. As there is a single instance of virtual base class which is shared by multiple classes that inherit from it, hence the constructor of the base class is only called once by the constructor of concrete class, which in our case is class D.

If there is any call for initializing the constructor of class A in class B or class C, while creating object of class D, all such calls will be skipped.

8.13. Summary.

In this chapter, we have learned the concept of OOPs. Also we have learned the concept of Single Inheritance, Multiple Inheritance, Multlevel Inheritance, Hierarchical inheritance, Hybrid Inheritance, Container Classes, , Virtual Base Classes, Construction in Derived classes, Virtual Function, Pure Virtual Functions, and also learned Abstract Classes.

8.13. Self Assessment Questions

1. How are data and functions organized in an object-oriented program?
2. What are the unique advantages of an object-oriented programming paradigm?
3. Distinguish between the following terms:

i. Inheritance and polymorphism

ii. Dynamic binding and message passing

4. Describe inheritance as applied to OOP.
5. What do you mean by dynamic binding? How it is useful in OOP?
6. What is a virtual base class ? When do we make it?
7. Write a program in c++ which demonstrate the use of inheritance.
8. What do you understand by function returning a pointer? Give any suitable example to support your answer.

Chapter 9

The C++ I/O System Basics

Objective:

At the end of this session students shall be learning:

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

Structure:

9.1 Introduction

9.2 C++ streams

9.3 C++ streams classes

9.4 Unformatted I/O Operations

9.5 Formatted console I/O Operations

9.6 Managing output with manipulators

9.7 Design Our Own Manipulators

9.8. Summary

9.9. Self Assessment Questions

9.1 Introduction:

C++ supports two complete I/O systems: the one inherited from C and the object- oriented I/O system defined by C++ (hereafter called simply the C++ I/O system). Like C-based I/O, C++'s I/O system is fully integrated. The different aspects of C++'s I/O system, such as console I/O and disk I/O, are actually just different perspectives on the same mechanism.

Every program takes some data as input and generates processed data as output following the input -process-output cycle. C++ supports all of C's rich set of I/O functions that can be used in the C++ programs. But these are restrained from using due to two reasons, first I/O methods in C++ supports the concept of OOP and secondly I/O methods in C can not handle the user defined data types such as class objects. C++ uses the concept of streams and stream classes to implement its I/O operation with the console and disk fils.

9.2 C++ streams:

A stream is a logical device that either produces or consumes information. A stream is linked to a physical device by the I/O system. All streams behave in the same way even though the actual physical devices they are connected to may differ substantially. Because all streams behave the same, the same I/O functions can operate on virtually any type of physical device. For example, one can use the same function that writes to a file to write to the printer or to the screen. The advantage to this approach is that you need learn only one I/O system.

A stream act like a source or destination, the source stream that provide data to the program is called the input stream and the destination stream that receive output from the program is called the output stream.

C++ containes cin and cout predefined streams that opens automatically when a program begins its execution.cin represents the input stream connected to the standard input device and cout represents the output stream connected to standard output device.

9.3 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 operstions with the console unit. These classes are declared in the header file iostrem. The file should be included in all programs that communicate with the console unit.

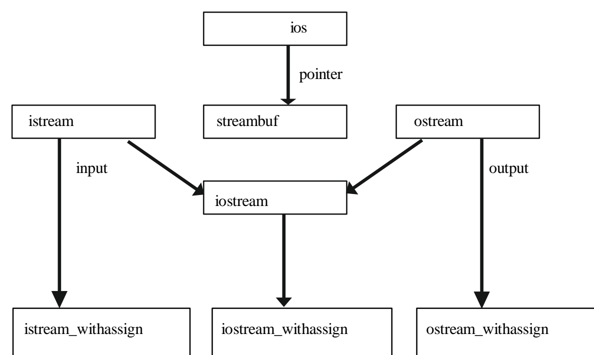


Figure 9.1 Stream classes for console I/O operations

As in figure 9.1 ios is the base class for istream(input stream) and ostream(output stream) which are base classes for iostream(input/output stream).The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream.

The class ios provides the basic support for formatted and unformatted input/output operations.

The class istream provides the facilities for formatted and unformatted input while the class ostream(through inheritance) provides the facilities for formatted output.

The class iostream provides the facilities for handling both input output streams. Three classes namely istream_with assign, ostream_withassign and iostream_with assign add assignment operators to these classes.

Table 9.1 Stream classes for console operations

Class name	Contents
ios(General input/output stream class)	Contains basic facilities that are ued 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

9.4 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 reads and display a line of text. The operator >> can be used to read a character but it will skip the white spaces and newline character.The above while loop will not work properly if the statement

cin >> c;

is used in place of cin.get (c);

The get(void) version is used as follows:

………….. char c;

c= cin.get();

…………

The value returned by the function get() is assigned to the variable c.

The function put(), a member of 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 variable ch.

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

cout.put(68);

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

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

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

The program 9.1 illustrate 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 getline() and write() functions:

A line of text can be read or display 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 invokedby 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 key board: Bjarne Stroustrup<press Return>

This input will be read correctly and assigned to the character array name. Let us suppose the input is as follows:

Object Oriented Programming<press Return>

In this case ,the input will be terminated after reading the following 19 characters Object Oriented Pro

Remember ,the two blank spaces contained in the string are also taken into

account.Strings cen be read using the operator >> as follows

cin>>name;

But remember cin can read strings that do not contain white spaces.This means that cin can read just one word and not a series of words such as “Bjarne Stroustrup”.But it can read the following string correctly:

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: Chennai

During fist run the newline character '\n\ at the end of “Delhi” which is waiting in the input queue is read by the getline() that follows immediately and therefore it does not wait for any response to the prompt 'enter city name again'.The character'\n' is read as an empty line.

The write () function displays an entire line and has the following form:

cout. write (line,size)

The first argument line represents the name of the string to be displayed and the second argument size indicates the number of characters automatically when the null character is encountered. If the size is greater than the length of line, then it displays beyond the bound of line.Program9.3 illustrates how write() method displays a string.

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

C++ Programming C++ Progr

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++ supports a number of features that could be used for formatting the output. These features include:

--ios class function and flags.

--manipulators.

--User-defined output functions.

The ios class contains a large number of member functions that would help us to format the output in a number of ways.The most importany ones among them 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

Manipulators are special functions that can be included in the I/O statements to alter the format parameter of stream .Table 9.3 shows some important manipulator functions that are frequently used. To access these manipulators, the file iomanip should be included in the program.

Table 9.3 Manipulators

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

In addition to these standard library manipulators we can create our own manipulator functions to provide any special output formats.

9.9.1 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

9.9.2 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;
```

The first two statement instruct :"print two digits after the decimal point in a field of five character width".Thus the output will be:

	1		2	3

Program 9.5 shows how the function width() and precision() are jointly used to control the output format.

Program 9.5: PRECISION SETTING WITH precision()

```
#include<iostream> #include<cmath> using namespac 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

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)

9.9.3 FILLING AND PADDING :fill()

The unused portion of field width are 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

Financial institutions and banks use this kind of padding while printing cheques so that no one can change the amount easily.Like precision (),fill()

Stays in effect till we change it. As shown in following program

Program 9.6

```
#include<iostream>

using   namespace   std;

int main()

{ cout.fill('<'); cout.precision(3); for(int n=1;n<=6;n++)

{

cout.width(5); cout<<n;

cout.width(10);

cout<<1.0/float(n)<<"\n";

if(n==3)

cout.fill('>');

}

cout<<"\nPadding   changed   \n\n";   cout.fill('#');   //fill()   reset   cout.width(15);

cout<<12.345678<<"\n"; return 0;

}
```

The output will be

```
<<<<1<<<<<<<<<1

<<<<2<<<<<<<<0.5

<<<<3<<<<<<0.333

>>>>4>>>>>>0.25
```

>>>>5>>>>>>>0.2

PADDING CHANGED

#########12.346

9.5.4 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 member of 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

9.6 Managing Output with Manipulators:

The header file iomanip provides a set of functions called manipulators which can be used to manipulate the output format. They provide the same features as that of the ios member functions and flags.

For example, two or more manipulators can be used as a chain in one statement as follows

```
cout<<manip1<<manip2<<manip3<<item;

cout<<manip1<<item1<<manip2<<item2;
```

This kind of concatenation is useful when we want to display several 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. The output can be made left-justified by modifying the statement follows:

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

One statement can be used to format output for two or more 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;
```

will print all the three values in one line with the field sizes of 5,10,15 respectively.

The following program illustrates the formatting of the output values using both manipulators and ios functions.

Program 9.7

```
#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";

double term,sum=0;

for (int n=1;n<=10;n++)

{

term=1.0/float(n);

sum=sum + term;

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

} return 0;

}
```

9.7 Designing our own manipulators:

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

```
{

……………

…......................................... (code)

……………..
```

```
return output;

}
```

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

Program 9.8

```
#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

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

9.8 Summary

A stream is a sequence of bytes and serves as a source or destination for an I/O data.

The source stream that provides data to the program is called as input stream and the destination stream that receives output from the program is called the output stream.

The C++ I/O system contains a hierarchy of stream classes used for input and output operations.These classes are declared in the header file 'iostream'.

cin represents the input stream connected to standard input device and cout represents the output stream connected to standard output device.

The >> operator is overloaded in the istream class as an extraction operator and the << operator is overloaded in the ostream class as an insertion operator.

We can read and write a line of text more efficiently using the line oriented I/O functions getline() and write() respectively.

The header file iomanip provides a set of manipulator functions to manipulate output formats.

9.9 Self Assessment Questions

9.1. What is a stream?

9.2. Describe briefly the features of I/O system supported by C++.

9.3. How is cout able to display various types of data without any special instructions?

9.4. Why it is necessary to include the file iostream in all our programs?

9.5. What is the role of iomanip file?

9.6. What is the basic difference between manipulators and ios member functions in implementation? Give examples.

Chapter 10
Working with Files

Objective:

At the end of this session students shall be learning:

- Introduction of Creating a Stream,
- Opening a File,
- Closing a File,
- Checking For Failure With File Commands,
- Detecting the End-of-file,
- file Pointers and their Manipulation,
- Reading/Writing a character From a File,
- Write()and read() Functions,
- Buffers and Sychronization,
- Other Functions,
- Random Access File Processing,
- Updating a File :Random Access,
- Command Line Arguments.

Structure:

10.1. Introduction Creating a Stream,

10.2. File Stream Classes

10.3. Steps of file operations

10.4. Finding End of file

10.5. File opening modes

10.6. File Pointers and their Manipulation,

10.7. Sequential Input and Output Operations

10.8. Error handling during file operations

10.9. Random Access

10.10. Summary.

10.11. Self Assessment Questions

10.1 Introduction of creating a Stream.

When a large amount of data is to be handled in such situations floppy disk or hard disk are needed to store the data .The data is stored in these devices using the concept of files. A file is a collection of related data stored in a particular area on a disk. Programs can be designed to perform the read and write operations on these files.

The I/O system of C++ handles file operations which are very much similar to the console input and output operations. It uses file streams as an interface between the programs and files. The stream that supplies data to the program is called input stream and the one that receives data from the program is called output stream. In other words input stream extracts data from the file and output stream inserts data to the file.

The input operation involves the creation of an input stream and linking it with the program and input file. Similarly, the output operation involves establishing an output stream with the necessary links with the program and output file.

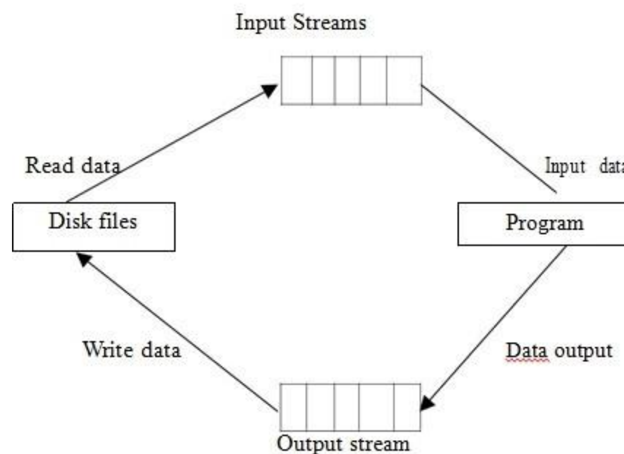


Fig. 10.1 File Input and output stream

10.2 File Stream Classes:

The I/O system of C++ contains a set of classes that defines the file handling methods. These include ifstream, ofstream and fstream.These classes are derived from fstreambase and form the corresponding iostream class. These classes
,designed to manage the disk files are declared in fstream and therefore this file is included in any program that uses files.

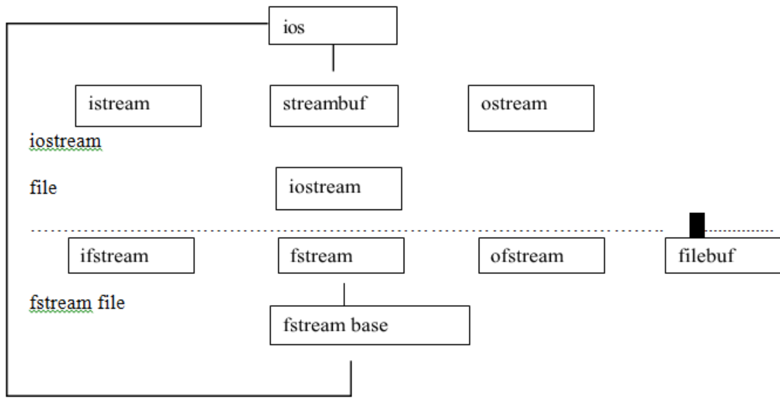


Fig. 10.2 Stream classes for file operations

10.3 Steps of File Operations:

For using a disk file the following things are necessary

1. Suitable name of file
2. Data type and structure
3. Purpose
4. Opening Method

Table 10.1 Detail of file stream classes

Class	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes.Also contain **close()** and **open()** as members.
fstreambase	Provides operations common to file streams.Serves as a base for **fstream,ifstream** and **ofstream** class.Contains **open()** and **close()** functions.
ifstream	Provides input operations.Contains **open()** with default input mode.Inherits the functions **get(),getline(),read(),seekg(),tellg()** functions from istream.
ofstream	Provides output operations.Contains **open()** with default output mode.Inherits **put(),seekp(),tellp()** and **write()** functions from **ostream**.
fstream	Provides support for simultaneous input and output operations.Contains open with default input mode.Inherits all the functions from **istream** and **ostream** classes through **iostream**.

The filename is a string of characters that makeup a valid filename for the operating system. It may contain two parts, primary name and optional period with extension.

Examples are Input.data, Test.doc etc. For opening a file firstly a file stream is created and then it is linked to the filename.A file stream can be defined using the classes ifstream, ofstream and fstream that contained in the header file fstream.The class to be used depends upon the purpose whether the write data or read data operation is to be performed on the file. A file can be opened in two ways:

(a) Using the constructor function of class.

(b) Using the member function open() of the class.

The first method is useful only when one file is used in the stream. The second method is used when multiple files are to be managed using one stream.

10.3.1 Opening Files using Constructor:

While using constructor for opening files, filename is used to initialize the file stream object. This involves the following steps

(i) Create a file stream object to manage the stream using the appropriate class i.e the class ofstream is used to create the output stream and the class create the input stream.

(ii) Initialize the file object using desired file name.

For example, the following statement opens a file named "results" for output:

of stream outfile("results"); //output only

This create outfile as an ofstream object that manages the output stream. Similarly, the following statement declares infile as an ifstream object and attaches it to the file data for reading (input).

If stream infile ("data");//input only

The same file name can be used for both reading and writing data.For example

Program1

……………………..

……………….

ofstream outfile ("salary");

//creates outfile and connects salary to it

………………

…………………..

Program 2

………………

……………

ifstream infile ("salary");

//creates infile and connects salary to it

………………..

………………….

The connection with a file is closed automatically when the stream object expires i.e when a program terminates. In the above statement ,when the program 1 is terminated, the salary file is disconnected from the outfile stream. The same thing happens when program 2 terminates.

Instead of using two programs, one for writing data and another for reading data ,a single program can be used to do both operations on a file.

…………

…………….

outfile.close(); //disconnect salary from outfile and connect to infile ifstream infile ("salary");

………….

…………… infile.close();

The following program uses a single file for both reading and writing the data .First it take data from the keyboard and writes it to file.After the writing is completed the file is closed.The program again opens the same file read the information already written to it and displays the same on the screen.

PROGRAM 10.1: WORKING WITH SINGLE FILE

```
//Creating files with constructor function #include <iostream.h>
#include <fstream.h> int main()
{
ofstream outf("ITEM"); cout <<"enter item name: "; char name[30];
cin >>name;
outf <<name <<"\n";
cout <<"enter item cost :"; float cost;
cin >>cost;
outf <<cost <<"\n"; outf.close(); ifstream inf("item"); inf >>name;
inf >>cost;


cout <<"\n";
cout <<"item name : " << name <<"\n"; cout <<"item cost: " << cost <<"\n"; inf.close();
return 0;
}
```

10.3.2 Opening Files using open()

The function open() can be used to open multiple files that uses the same stream object. For example to process a set of files sequentially,in such case a single stream object can be created and can be used to open each file in turn. This can be done as follows;

File-stream-class stream-object; stream-object.open ("filename");

The following example shows how to work simultaneously with multiple files

PROGRAM 10.2: WORKING WITH MULTIPLE FILES

```
//Creating files with open() function #include <iostream.h> #include<fstream.h>
int main()
{
ofstream fout; fout.open("country"); fout<<"United states of America \n"; fout<<"United Kingdom"; fout<<"South korea";
fout.close(); fout.open("capital"); fout<<"Washington\n"; fout<<"London\n"; fout<<"Seoul \n"; fout.close();
const int N =80; char line[N];
```

```
ifstream fin;

fin.open("country"); cout<<"contents of country file \n"; while (fin)
{
fin.getline(line,N); cout<<line;
}
fin.close(); fin.open("capital");
cout<<"contents of capital file"; while(fin)
{
fin.getline(line,N); cout<<line;
}
fin.close(); return 0;
}
```

10.4 Finding End of File:

While reading a data from a file,it is necessary to find where the file ends i.e end of file.The programmer cannot predict the end of file,if the program does not detect end of file,the program drops in an infinite loop.To avoid this,it is necessary to provide correct instruction to the program that detects the end of file.Thus when end of file of file is detected,the process of reading data can be easily terminated. An ifstream object such as fin returns a value of 0 if any error occurs in the file operation including the end-of – file condition.Thus the while loop terminates when fin returns a value of zero on reaching the end-of –file condition.There is an another approach to detect the end of file condition.

The statement if(fin1.eof() !=0)

```
{
exit(1);
}
```

returns a non zero value if end of file condition is encountered and zero otherwise.Therefore the above statement terminates the program on reaching the end of file.

10.5 File Opening Modes:

The ifstream and ofstream constructors and the function open() are used to open the files.Upto now a single arguments a single argument is used that is filename.However,these functions can take two arguments, the second one for specifying the file mode.The general form of function open() with two arguments is:

stream-object.open("filename",mode);

The second argument mode specifies the purpose for which the file is opened. The prototype of these class member functions contain default values for second argument and therefore they use the default values in the absence of actual values. The default values are as follows :

ios:: in   for ifstream functions meaning open for reading only. ios::out for ofstream functions meaning open for writing only.

The file mode parameter can take one of such constants defined in class ios.The following table lists the file mode parameter and their meanings.

Table 10. File Mode Operation

Parameter	**Meaning**
ios::app	Append to end-of-file
ios::ate	Go to end-of-file on opening
ios::binary	Binary file
ios::in	Open file for reading only
ios::nocreate	Open fails if file the file does not exist
ios::noreplace	Open fails if the file already exists
ios::out	Open file for writing only
ios::trunk	Delete the contents of the file if it exists

10.6 File Pointers and Manipulators:

Each file has two pointers known as file pointers,one is called the input pointer and the other is called output pointer..The input pointer is used for reading the contents of of a given file location and the output pointer is used for writing to a given file location. Each time an input or output operation takes place,the appropriate pointer is automatically advanced.

10.6.1 Default actions:

When a file is opened in read-only mode,the input pointer is automatically set at the beginning so that file can be read from the start.Similarly when a file is opened in write-only mode the existing contents are deleted and the output pointer is set at the beginning. This enables us to write the file from start.In case an existing file is to be opened in order to add more data, the file is opened in 'append' mode.This moves the pointer to the end of file.

10.6.2 Functions for Manipulations of File pointer:

All the actions on the file pointers takes place by default. For controlling the movement of file pointers file stream classes support the following functions

- seekg() Moves get pointer (input)to a specified location.
- seekp() Moves put pointer (output) to a specified location.
- tellg() Give the current position of the get pointer.
- tellp() Give the current position of the put pointer.

For example, the statement infile. seekg(10);

moves the pointer to the byte number 10.The bytes in a file are numbered beginning from zero.Therefore ,the pointer to the 11th byte in the file.Consider the following statements:

ofstream fileout; fileout.open("hello",ios::app); int p=fileout.tellp();

On execution of these statements,the output pointer is moved to the end of file "hello" And the value of p will represent the number of bytes in the file.

10.10.3 Specifying the Offset:

'Seek' functions seekg() and seekp() can also be used with two arguments as follows: seekg (offset,refposition);

seekp (offset,refposition);

The parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter reposition.

The reposition takes one of the following three constants defined in the ios class:

ios::beg

Start of file

ios::cur

Current position of the pointer ios::end   End of file

The seekg() function moves the associated file's 'get' pointer while the seekp() function moves the associated file's 'put 'pointer.The following table shows some sample pointer offset calls and their actions.fout is an ofstream object.

Table 10.3 Pointer offset calls

Seek call	**Action**
fout.seekg(o,ios::beg)	Go to start
fout.seekg(o,ios::cur)	Stay at the current position
fout.seekg(o,ios::end)	Go to the end of file
fout.seekg(m,ios::beg)	Move to (m+1)th byte in the file
fout.seekg(m,ios::cur)	Go forward by m byte from current position
fout.seekg(-m,ios::cur)	Go backward by m bytes from current position.
fout.seekg(-m,ios::end)	Go backward by m bytes from the end

10.7 Sequential Input and Output Operations:

The file stream classes support a number of member functions for performing the input and output operations on files.One pairs of functions,put() and get() are designed for handling a single character at a time.Another pair of functions, write(),read() are designed to write and read blocks of binary data.

10.7.1 put() and get() Functions:

The function put() writes a single character to the associated stream.Similarly,the function get() reads a single character from the associated stream.The following program illustrates how the functions work on a file.The program requests for a string.On receiving the string,the program writes it,character by character,to the file using the put() function in a for loop.The length of string is used to terminate the for loop.

The program then displays the contents of file on the screen.It uses the function get() to fetch a character from the file and continues to do so until the end –of –file condition is reached.The character read from the files is displayed on screen using the operator <<.

PROGRAM 10.3: I/O OPERATIONS ON CHARACTERS

```
#include <iostream.h> #include <fstream.h> #include<string.h>
int main()
{
char string[80]; cout<<"enter a string \n"; cin>>string;
int len =strlen(string); fstream file;
file.open("TEXT". Ios::in | ios::out);

for (int i=o;i<len;i++) file.put(string[i]);   file .seekg(0);
char ch; while(file)
{
file.get(ch); cout<<ch;
}
return 0;
}
```

10.7.2 write() and read () functions:

The functions write() and read(),unlike the functions put() and get(), handle the data in binary form. This means that the values are stored in the disk file in same format in which they are stored in the internal memory.An int character takes two bytes to store its value in the binary form, irrespective of its size. But a 4 digit int will take four bytes to store it in the character form. The binary input and output functions takes the following form:

```
infile.read (( char  ) & V,sizeof (V));

outfile.write (( char ) & V ,sizeof (V));
```

These functions take two arguments.The first is the address of the variable V, and the second is the length of that variable in bytes.The address of the variable must be cast to type char(i.e pointer to character type).The following program illustrates how these two functions are used to save an array of floats numbers and then recover them

for display on the screen.

PROGRAM 10.4:

```
// I/O OPERATIONS ON BINARY FILES
#include        <iostream.h>
#include         <fstream.h>
#include <iomanip.h>
const char  filename =”Binary”; int
main()
float height[4] ={ 175.5,153.0,167.25,160.70};
ofstream outfile; outfile.open(filename);
outfile.write((char        )        &        height,sizeof(height));
outfile.close();
for          (int          i=0;i<4;i++)
height[i]=0; ifstream infile;
infile.open(filename);
infile.read    ((char    )    &    height,sizeof    (height));    for
(i=0;i<4;i++)
{
cout.setf(ios::showpoint); cout<<setw(10)<<setprecision(2)<<height[i];
}
infile.close(); return 0;
}
```

10.8 Error Handling during File Operations:

There are many problems encounterd while dealing with files like

- a file which we are attempting to open for reading does not exist.
- The file name used for a new file may already exist.
- We are attempting an invalid operation such as reading past the end of file.
- There may not be any space in the disk for storing more data.
- We may use invalid file name.
- We may attempt to perform an operation when the file is not opened for that purpose.

The C++ file stream inherits a 'stream-state 'member from the class ios. This member records information on the status of a file that is being currently used. The stream state member uses bit fields to store the status of error conditions stated above. The class ios support several member functions that can be used to read the status recorded in a file stream.

Table 10.4 Error Handling Functions

Function	**Return value and meaning**
eof()	Returns true(non zero value) if end of file is encountered while reading otherwise returns false(zero).
fail()	Returns true when an input or output operation has failed .
bad()	Returns true if an invalid operation is attempted or any unrecoverable error has occurred.However,if it is false,it may be possible to recover from any other error reported and continues operation.
good()	Returns true if no error has occurred.This means all the above functions are false.For instance,if file.good() is true.all is well with the stream file and we can proceed to perform I/O operations.When it returns false,no further operations is carried out.

These functions can be used at the appropriate places in a program to locate the status of a file stream and thereby to take the necessary corrective measures. Example:

```
……………
…………..
ifstream infile;infile.open(“ABC”); while(!infile.fail())
{
…………
…................................(process the file)
…………….
}
if (infile.eof())
{
…................................. (terminate the program normally)
}
else
```

```
if (infile.bad())

{
…......................................... (report fatal error)
}
else
{
infile.clear(); //clear error state

……….
……….

}
……..
………..
```

The function clear() resets the error state so that further operations can be attempted.

10.9 Command Line Arguments:-

Like C,C++ also support the feature of command line argument i.e passing the arguments at the time of invoking the program.They are typically used to pass the names of data files. Example:

```
C>exam data results
```

Here exam is the name of file containing the program to be executed and data and results are the filenames passed to program as command line arguments. The command line arguments are typed by the user and are delimited by a space. The first argument is always the filename and contains the program to be executed. The main() functions which have been using up to now without any argument can take two arguments as shown below:

```
main(int argc,char  argv[])
```

The first argument argc repesents the number of arguments in command line. The second

argument argv is an array of character type pointers that points to the the command line arguments.The size of this array will be equal to the value of argc.

For instance, for command line C>exam data results

The value of argc would be 3 and the argv would be an array of three pointers to string as shown:

argv[0] exam argv[1] data argv[2] results

The argv[0] alwayas represents the command name that invokes the program.The character pointer argv[1], and argv[2] can be used as file names in the file opening statements as shown:

…………

…………

inline.open(argv[1]); //open data file for reading

…………..

…………..

outfile.open(argv[2]); //open result file for writing

…………..

…………..

10.9 Random Access:

Every file maintains two pointers called get_pointer (in input mode file) and put_pointer (in output mode file) which tells the current position in the file where reading or writing will

takes place. (A file pointer in this context is not like a C++ pointer but it works like a book-mark in a book.). These pointers help attain random access in file. That means moving directly to any location in the file instead of moving through it sequentially.

There may be situations where random access in the best choice. For example, if you have to modify a value in record no 20, then using random access techniques, you can place the file pointer at the beginning of record 20 and then straight-way process the record. If sequential access is used, then you'll have to unnecessarily go through first twenty records in order to reach at record 20.

10.9.1 The seekg(), seekp(), tellg() and tellp() Functions:

In C++, random access is achieved by manipulating seekg(), seekp(), tellg() and tellp() functions. The seekg() and tellg() functions allow you to set and examine the get_pointer, and the seekp() and tellp() functions perform these operations on the put_pointer.

The seekg() and tellg() functions are for input streams (ifstream) and seekp() and tellp() functions are for output streams (ofstream). However, if you use them with an fstream object then tellg() and tellp() return the same value. Also seekg() and seekp() work the same way in an fstream object. The most common forms of these functions are:

	istream & seekg(long);	Form 1
seekg()	istream & seekg(long, seek_dir);	Form 2
seekp()	ofstream & seekp(long);	Form 1
	ofstream & seekp(long, seek_dir);	Form 2
tellg()	long tellg()	
tellp()	long tellp()	

The working of seekg() & seekp() and tellg() & tellp() is just the same except that seekg() and tellg() work for ifstream objects and seekp() and tellp() work for ofstream objects. In the above table, seek_dir takes the definition enum seek_dir { beg, cur, end};.

The seekg() or seekp(), when used according to Form 1, then it moves the get_pointer or put_pointer to an absolute position. Here is an example:

```
ifstream fin; ofstream fout;
:// file opening routine
fin.seekg(30);   // will move the get_pointer (in ifstream) to byte number 30 in the file
fout.seekp(30);  // will move the put_pointer (in ofstream) to byte number 30 in the file
```

When seekg() or seekp() function is used according to Form 2, then it moves the get_pointer or put_pointer to a position relative to the current position, following the definition of seek_dir. Since, seek_dir is an enumeration defined in the header file iostream.h, that has the following values:

ios::beg, // refers to the beginning of the file ios::cur, // refers to the current position in the file ios::end} // refers to the end of the file

Here is an example.

```
fin.seekg(30, ios::beg);      // go to byte no. 30 from beginning of file linked with fin
fin.seekg(-2, ios::cur); // back up 2 bytes from the current position of get pointer
fin.seekg(0, ios::end);// go to the end of the file
fin.seekg(-4, ios::end); // backup 4 bytes from the end of the file
```

The functions tellg() and tellp() return the position, in terms of byte number, of put pointer and get_pointer respectively, in an output file and input file.

C++ File Pointers and Random Access Example

Here is an example program demonstrating the concept of file pointers and random access in a C++ program:

```
/ C++ File Pointers and Random Access
*        This program demonstrates the concept
*        of file pointers and random access in
*        C++    /    #include<fstream.h>    #include<conio.h>    #include<stdlib.h>
#include<stdio.h> #include<string.h> class student
{

int rollno;
char name[20]; char branch[3]; float marks; char grade; public:
void getdata()
{
cout<<"Rollno: "; cin>>rollno; cout<<"Name: "; cin>>name; cout<<"Branch: ";


cin>>branch;
```

```
cout<<"Marks: "; cin>>marks; if(marks>=75)

{

grade = 'A';
}
else if(marks>=60)
{
grade = 'B';
}
else if(marks>=50)
{
grade = 'C';
}
else if(marks>=40)
{

}
else
{

}
grade = 'D';

grade = 'F';


void putdata()
{
cout<<"Rollno: "<<rollno<<"\tName: "<<name<<"\n"; cout<<"Marks: "<<marks<<"\tGrade:
"<<grade<<"\n";
}
```

```
int getrno()
{
return rollno;
}
void modify();

}stud1, stud;
void student::modify()
{

cout<<"Rollno: "<<rollno<<"\n";
cout<<"Name: "<<name<<"\tBranch: "<<branch<<"\tMarks: "<<marks<<"\n";
cout<<"Enter new details.\n"; char nam[20]=" ", br[3]=" "; float mks;
cout<<"New name:(Enter '.' to retain old one): "; cin>>nam;
cout<<"New branch:(Press '.' to retain old one): "; cin>>br;
cout<<"New marks:(Press -1 to retain old one): "; cin>>mks;
if(strcmp(nam, ".")!=0)
{
strcpy(name, nam);
}
if(strcmp(br, ".")!=0)
{
strcpy(branch, br);
}
if(mks != -1)
{
marks = mks; if(marks>=75)
{
grade = 'A';
}
else if(marks>=60)
{
grade = 'B';
}
```

```
else if(marks>=50)

{
grade = 'C';
}

else if(marks>=40)
{
}
else
{

}
}
}
grade = 'D';

grade = 'F';

void main()
{
clrscr();
fstream fio("marks.dat", ios::in | ios::out); char ans='y';
while(ans=='y' || ans=='Y')
{
stud1.getdata();
fio.write((char )&stud1, sizeof(stud1)); cout<<"Record added to the file\n"; cout<<"\nWant to
enter more ? (y/n).."; cin>>ans;
}
clrscr(); int rno; long pos;
char found='f';
cout<<"Enter rollno of student whose record is to be modified: "; cin>>rno;
```

```
fio.seekg(0);
while(!fio.eof())

{
pos = fio.tellg();
fio.read((char )&stud1, sizeof(stud1)); if(stud1.getrno() == rno)
{

stud1.modify(); fio.seekg(pos);
fio.write((char )&stud1, sizeof(stud1)); found = 't';
break;
}
}
if(found=='f')
{
cout<<"\nRecord not found in the file..!!\n"; cout<<"Press any key to exit...\n";
getch();
exit(2);
}

fio.seekg(0);
cout<<"Now the file contains:\n"; while(!fio.eof())
{

}
fio.close();
getch();

fio.read((char )&stud, sizeof(stud)); stud.putdata();
```

Here are the sample runs of the above C++ program:

10.10. Summary:

Stream is nothing but flow of data .In object oriented programming the streams are controlled

using classes. The istream and ostream classes control input and output functions respectively. The iostream class is also a derived class .It is derived from istream and ostream classes. There are three more derived classes istream_withassign,ostream_withassign and iostream_withassign.They are derived from istream,ostream and iostream respectively. .There are two methods constructor of class and member function open() of the class for opening the file. The class ostream creates output stream objects and ifstream creates input stream objects. The close() member function closes the file. When end of file is detected the process of readind data can be easily terminated.The eof() function is used for this purpose.The eof() stands for end of file.The eof() function returns 1 when end of file is detected. The seekg () functions shifts the associated file's input file pointer and output file pointer. The put() and get() functions are used for reading and writing a single character whereas write() and read() are used to read or write block of binary data.

In C++, random access is achieved by manipulating seekg(), seekp(), tellg() and tellp() functions. The seekg() and tellg() functions allow you to set and examine the get_pointer, and the seekp() and tellp() functions perform these operations on the put_pointer.

10.11. Self Assessment Questions:

1. What are input and output streams?
2. What are the various classes available for file operations.
3. What is a file mode ?describe the various file mode options available.
4. Describes the various approaches by which we can detect the end of file condition.
5. What do you mean by command line arguments?

Chapter 11
Template

Objective:

At the end of this session students shall be learning:

- Introduction,
- Generic Functions,
- A Function with Two Generic Data Types,
- Explicitly Overloading a generic Function,
- Overloading Function Templates,
- using Standard Parameters with Template Function,
- Generic Functions Restrictions,
- Generic Class,
- Using Default Arguments with Template Classes,
- Template Parameters,
- Template Specialization,
- The Typename and Export Keywords

Structure:

11.1 Introduction

11.2 Class templates

11.3 Multiple parameters in class templates

11.4 Function templates

11.5 Multiple parameters in function templates

11.6 Overloading of template functions

11.7 Member function templates

11.8 Non-type template arguments

11.9. Summary

11.10. Self Assessment Questions

11.1 Introduction

Template is a new concept which enables us to define generic and functions and thus provides support for generic programming. Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.

A template can be used to create a family of classes or functions. For example, a class template for an array class would enable us to create arrays of various data types such as int array and float array .similarly, we can define a template for a function, say mul(),hat would help us create versions of mul() for multiplying int, float and double type values.

A template can be considered as a kind of macro. When an object of a specific type is define for actual use, the template definition for that class is substitute with the required data type. Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized class or functions.

11.2 Class Template:

Consider a vector class defined as follows: Class vector

```
{
int v ; int size; public:
vector(int m )    // create a null vector
{
v=new int[size = m]; for(int i=0;i<size;i++) v[i]=0;
}
vector(int a)     //create a vector from an array
{
for(int i=0;i<size;i++) v[i]=a[i];
}

Template<class T> class classname
{
…
//class member specification
//with anonymous type T
```

```
//whenever appropriate
….
};
```

The template definition of vector class shown below illutrates the syntax of a template:

```
template<class T> class vector
{
T v;                    // type T vector int size;
public: vector(int m )
{
v=new T [size = m]; for(int i=0; i<size; i++) v[i] =0;
}
vector (T a)
{
for(int i=0;i<size; i++) v[i]=a[i];
}
T operator( vector &y)
{
T sum =0;
for(int i=0;i<size;i++) sum+=this->v[i]y-v[i]; return sum;}
};
```

Remember:

The class template definition is very similar to an ordinary class definition except the prefix template<class T> and the use of type T. This prefix tells the complier that we are going to declare a template and use T as a type name in the Declaration. Thus, vector has become a parameterized class with the type T as its parameters. T may be substituted by any data type including the user defined types.

Now we can create vectors for holding different data types.

Example;

```
                 v1(10        //10    element    int
vector<int>    );            vector
                 v2(30        //30    element    float
vector<float> );             vector
```

The type **T** may represent a class name as well.

Example:

```
Vector<complex> v3 (5);
                              // vector of 5 complex numbers
```

A class created from a class template is called a template class. The syntax for defining an object of a template class is:

Classname<type> objectname (arglist);

This process of creating a specific class from a class template is called instantiation. The complier will perform the error analysis only when an instantiating take place. It is, therefore, advisable to create and debug an ordinary class before converting it in to template.

Example of class template #include <iostream> using namespace std; const size=3;

```
template<class T>
class vector
{
Tv; public: vector()
{
v=new T[size];   for(int i=0;i<size;i++) v[i]=0;
}

int operator9vector &y) //scalar product
{
int sum=0;
for(int i=0;i<size;i++) sum+=this->v[i]y-v[i]; return sum;
}
};
```

The vector class can store an array of int numbers and perform the scalar product of two int vector as shown below:

```
int main()
{
int x[3]={1,2,3};
int y[3]={4,5,6};
vector v1(3);    //create a null vector of 3 integers vector v2(3);
v1=x;   //create v1 from the array x v2=y;
int R=v1v2; cout<<"R="r; return 0;
}
```

Now suppose we want to define a vector that can store an array of float value. We can do this simply replacing the appropriate int declaration with float in the vector class. This means that we can have to redefine the entire class all over again.

Assume that we want to define a vector class with the data type as a parameter and then use this class to create a vector of any data type instead of defining a new class every time. The template mechanism enables us to achieve this goal.

As mentioned earlier, template allows us to define generic classes. It is simple process to create a generic class using a template with an anonymous type. The general format of a class template is:

```
vector(T a)
{
for(int i=0;i<size;i++) v[i]=a[i];
}
T operator(vector &y)
{
T sum=0;
for(int i=0;i<size;i++) sum+=this->v[i]y.v[i]; return sum;
}
};
int main()
{
int x[3]={1,2,3};
int y[3]={4,5,6};
vector<int> v1; vector<int> v2; v1=x;
v2=y;
int R= v1v2; cout<<"R="<<r<<"\n"; return 0;
}
```

The output would be: R=32

Another Example: #include <iostream> using namespace std; const size=3; template<classT> class vector

```
{
Tv; public:
vector ()
{
v=new T[size]; for(inti=0;i<size;i++) v[i]=0;
}
vector(T a)
{
for(int i=0;i<size;i++) v[i]=a[i];
}
T operator(vector &y)
{
T sum=0; for(int=0;i<size;i++) sum+=this->v[i]y.v[i]; return sum;
}
};
Int main()
{
float x[3]={1.1,2.2,3.3};
float y[3]={4.4,5.5,6.6};
vector <float> v1; vector <float>v2; v1=x;
v2=y;
float R=v1v2; cout<<"R="<<R<<"\n"; return 0;
}
```

The output would be:

R=311.720001

11.3 Class Templates with Multiple Parameters

We can use more than one generic data type in a class. They are declared as a comma- separated list within the template specification as shown below:

```
Template<class T1, class t2, …..> class classname
{
```

```
……
……
…….
}
;
```

Example with two generic data types: #include <iostream>

```
using name space std; template<class t1,class t2> class Test
{
T1 a;
T2 b;
public:
test(T1 x, T2 y)
{
a=x; b=y;
}
void show()
{
cout<<a<<"and"<<<<"\n";
}
};
int main()

{
Test<float,int> test1(1.23,123); Test<int,char> test2(100,'W'); test1.show();
test2.show(); return 0;
};
```

Output would be:

1.23 and 123

100 and W

11.4 Function Templates

Like class templates, we can also define function templates that cold be used to create a family of functions with different argument types. The general format of a function template is:

```
template<class T>
returntype functionname (argument of type T)
{
//
//body of function
//with Type T
//whenever appropriate
//……………
}
```

The function template syntax is similar to that of the class template except that we are defining functions instead of classes. We must use the template parameter T as and when necessary in the function body and its argument list.

The following example declares a swap () function template that will swap two values of a given type of data.

```
template <class T> void swap(T&x , T&y)
{
T temp =x; x=y; y=temp;
}
```

This essential declares a set of overloading functions, one for each type of data. We can invoke the swap () function likes any ordinary function .for example, we can apply the swap () function as follows:

```
void f ( int m , int n , float b )
{
swap ( m , n); //swap two integer values swap ( a , b);     //swap two float values
//……..
}
```

This will generate a swap () function template for each set of argument types. Example will show how a template function is defined and implemented.

An example:

```
#include <iostream> using namespace std; Template <class T> void swap (T &x, T &y)
{
T         temp=x; x=y;
```

```
y=temp;
}

void fun( int m, int n, float a, float b)
{
cout<<"m and n before swap:"<<m<<""<<n<<"\n"; swap(m,n);
cout<<"m and n after swap:"<<m<<""<<n<<"\n"; cout<<"a and b before swap:"<<a<<""<<b<<"\n"; swap(a,b);
cout<<"a and b after swap:"<<a<<""<<b<<"\n";
}
int main ()

{ fun(100,200,11.22,33.44);
return 0;
}
```

Output would be:

```
m and n before swap :        100    200
m and n after swap :         200    100
a and b before swap:       11.22    33.439999
a and b after swap :       33.439999 11.22
```

Another function often used is sort () for sorting arrays of various types such as int and double. The following shows a function template for bubble sort:

```
Template<class T> bubble( T a[] , int n )
{
for( int i=0 ; i<n-1 ; i++ ) for( int j=n-1 ; i<j ; j-- )
if ( a[j] < a[j-1] )
{
T temp = v[j]; v[j]=v[j-1]; v[j-1]=temp;
}
}
```

Note that the swapping statements T temp = v[j];
v[j]=v[j-1]; v[j-1]=temp;
May be replaced by the statement swap(v[j], v[j-1]);

Where swap () has been defined as a function template. Here is another example where a function returns a value. template<class T>

```
T max ( T x, T y)
{
return x>y ? x : y;
}
```

A function generated from a function is called a template function. Program of Bubble Sort demonstrates the use of two template functions in nested from for implementing the bubble sort algorithm.

```
#include <iostream> using namespace std ; template<class T> void bubble (T a[], int n)
{
for (int i=0; i<n-1; i++)
{
for (int j=n-1; i<j; j--)
i f (a[j] < a[j-1])
{
swap (a [j],a[j-1];// calls template function
}
}
templte<class X>
void swap(X &a, X &b)
{
x        temp=a; a= b; b= temp;
}
int main()
{ int x[5] = { 10,50,30,40,20,};
float y[5] ={1.1,5.5,3.3,4.4,2.2,};
bubble ( x , 5 ); // calls template function for int values
bubble ( y , 5 ); // calls template function for floate values
cout << "sorted x-array:"; for ( int i=0; i<5; i + + ) cout << x[i] << " " ;
cout << endl ;
```

```
cout     <<  " Sorted y-array : " ; for (int                    j=0;      j<5;      j + +) cout        << y[j]
         << " ";
cout     << endl          ;

return   0;
}
```

The output would be:

sorted x-array: 10 20 30 40 50

sorted y- array: 1.1 2.2 3.3 4.4 5.5 Another example:

```
#include <isotream> #include  <iomanip> #include <cmath> using namespace std; template
<class T>
void rootes(T  a,T        b,T       c)
{
T         d  = bb - 4ac;
if (d= =   0)        // Roots are equal
{
cout <<  "R1 =  R2  =  " <<       -b/(2a)  <<  endl;
}
else if (d>0)       //two real roots
{
cout<<"roots are real \n"; float R =sqrt (d);
float R1 = (-b+R)/(2a); float R2 = ( -b+R )/(2a);
cout<< "R1 = "<< R1 << " and"; cout <<R2 = "<< R2 << endl;
}
else      // roots are complex
{
cout <<"roots are complex \n"; float R1 = -b/( 2a);
float R2 = sqrt( -d )/( 2a );
cout <<" real part = " << R1 << endl; cout<< "imaginary part =" << R2; cout<< endl;
}


}
int main()
```

```
{
cout<< "integer coefficients \n"; roots(1,-5 ,6);
cout << "\n float coefficients \n"; roots (1.5, 3.6, 5.0);
return 0;
}
```

Output would be :

integer coefficients roots are real

R1= 3 and R2 =2

float coefficients roots are complex

real part = -1.2 imaginary part = 1.3757985

11.5 Function Template with Multiple Parameters

Like template classes, we can use more than one generic data type in the template statement, using a comma-separated list shown below:

```
template<class T1 , class T2, …..>
returntype functionname(arguments of types T1, T2, …)
{
……
……
……
}
```

Example with two generic types in template functions:

```
#inlude <iostream>
#include<string> using
namespace std;
template<class T1,class T2>
void display( T1 x, T2 y)

{
cout<<x<<" "<<y<<"\n";
}
int main()
```

```
{
display(1999, "EBG");
display(12.34, "1234);
return 0;
}
```

The output would be:

1999 EBG

12.34 1234

11.6 Overloading of Template Functions:

A template function may be overloaded either by template functions or ordinary functions of its name. In such cases, the overloading resolution is accomplished as follows:

1. Call an ordinary function that has an exact match.
2. Call a template function could be created with an exact match.
3. Try normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match is found. Note that no automatic conversions are applied to arguments on the template functions. Example for showing how a template function is overloaded with an explicit function.

```
#include <iostream>
#include <string> using
namespace std; template
<class T> void
display(T x)
{
cout<<"template display:" << x<< "\n";
}
void display ( int x)
{
cout<<"Explicit display: "<< x <<"\n";
}
```

int main()

{

display(100); display(12.34);

display('c'); return 0;

}

The output would be:

Explict display:100 template

display:12.34 template display:c

Remember:

The call display (100) invokes the ordinary version of display() and not the template version.

11.7 Member Function Templates:

When we create a class template for vector, all the member functions were defined as inline which was not necessary. We would have defined them outside the class as well. But remember that the member functions of the template classes themselves are parameterized by the type argument and therefore these functions must be defined by the function templates. It takes the following general form:

```
Template <class T>
returntype classname <T> :: functionname(arglist)
{
……
……..
…….
}
```

The vector class template and its member fnctions are redefined as follow:

```
// class template……….
template<class T>
class vector
{
Tv; int size; public:

vector(int m); vector(T
a); T operator(vector &
y);
```

```
};
//member function templates
template<class T> vector<T>
:: vector (int m );
{
v=new       T[size=m];
for(int i=0; i<size ; i++)
v[i]= 0;
}
template<class T> vector
<T>::vector(ta)
{
for(int i=0; i<size ; i++)
v[i]=a[i];
}
template< class T >
T vector < T > :: operator(vector & y)
{
T sum =0;
for ( int i=0; i< size ; i++) sum
+=  this  ->  v[i]y.v[i];  return
sum;
}
```

11.8 Non-Type Template Arguments

We have seen that a template can have multiple arguments. It is also possible to use non-type arguments. That is, in addition to the type argument T, we can also use other arguments such as strings, function names, constant expressions and built-in types.

Consider the following example: Template<class T, int size> Class array

```
{
T a[size];//automatic array initialization
//…………
//………..
};
```

This template supplies the size of the array as an argument. This implies that the size of the array is known to the complier at the compile time itself. The arguments must be specified whenever a template class is created. Example:

```
Array <int,10> a1; Array <float,5> a2; Array <char,20> a3;
//array of 10 integers
//array of 5 floats
//string of size 20
```

11.9. Summary

C++ supports a mechanism known as template to implement the concept of generic programming. Template allows us to generate a family of classes or a family of functions to handle different data types. Template classes and functions eliminate code duplication for different types and thus make the program development easier and more manageable. We can use multiple parameters in both the class templates and function templates. A specific class created from a class template is called a template class and the process of creating a template class is known as instantiation. Like other functions, template functions can be overloaded. Member function of a class template must be defined as function templates using the parameters of the class template. We may also use non type parameters such basic or derived data types as arguments templates.

11.10. Self Assessment Questions

1. What is generic programming? How is it implemented in c++?
2. A template can be considered as a kind of macro. Then, what is the difference between them?
3. Distinguish between overloaded functions and function templates.
4. Distinguish between the term class template and template class.
5. A class template is known as a parameterized class comment.
6. Write a function template for finding the minimum value contained in an array.
7. Write a class template to represent a generic vector. Include member functions to

perform the following tasks:

(a) To create the vector

(b) To modify the value of a given element

(c) To multiply by a scalar values

(d) To display the vector in the form (10, 20, 30 …)

Chapter 12

Exception Handling

Objective:

At the end of this session students shall be learning:

- Introduction,
- The STI Programming Models,
- Containers,
- Algorithms,
- Iterators,
- Function Objects,
- Allacators,
- Adaptors

Structure:

12.1 Introduction

12.2 Principles of Exception handling

12.3 Throwing mechanism

12.4 Catching mechanism

12.5 Re-throwing an Exception

12.6 Specifying Exception

12.7 Summary

12.8 Self Assessment Questions

12.1 Introduction:

Usually there are mainly two type of bugs, logical errors and syntactic errors. The logical errors occur due to poor understanding of problem and syntactic errors arise due to poor understanding of language. There are some other problems called exceptions that are run time anomalies or unused conditions that a program may encounter while executing. These anomalies can be division by zero,access to an array outside of its bounds or running out of memory or disk space. When a program encounters an exceptional condition it is important to identify it and dealt with it effectively. An exception is an object that is sent from the part of the program where an error occurs to that part of program which is going to control the error.

12.2 Principles of Exception handling:-

Exceptions are basically of two types namely, synchronous and asynchronous exceptions. Errors such as "out of range index" and "over flow" belongs to synchronous type exceptions. The errors that are caused by the events beyond the control of program (such as keyboard interrupts) are called asynchronous exceptions.

The purpose of exception handling mechanism is to detect and report exceptional circumstances so that appropriate action can be taken. The mechanism for exception handling is

1. Find the problem (hit the exception).
2. Inform that an error has occurred (throw the exception).
3. Receive the error information (Catch the exception).
4. Take corrective actions (Handle the exception).

The error handling code mainly consists of two segments, one to detect error and throw exceptions and other to catch the exceptions and to take appropriate actions.

12.3 Exception handling mechanism:-

C++ exception handling mechanism is basically built upon three keywords namely try, throw and catch. The keyword try is used to preface a block of statements which may generate exceptions. This block of statement is called try block. When an exception is detected it is thrown using throw statement in the try block. A catch block defined by the keyword catch 'catches' the exception thrown by the throw statement in the try block and handles it appropriately. The catch block that catches an exception must immediately follow the try block that throws the exception. The general form for this is

```
……………….
……………….. try
{
…………

……………..
//block of statements which detects and throw an exceptions
throw exception;

…………….
…………….
}
catch(type arg)
```

```
//catches exceptions
{
…...............................................// Block of statements that handles the exceptions
………………
…………….
}
………….
…………..
```

When the try block throws an exception, the program control leaves the try block and enters the catch statement of the catch block. If the type of object thrown matches the arg type in the catch statement, then the catch block is executed for handling the exception. If they do not match, the program is aborted with the help of abort() function which is executed implicitly by the compiler. When no exception is detected and thrown, the control goes to the statement immediately after the catch block i.e catch block is skipped. The below diagram 12.1 will show the mechanism of exception handling

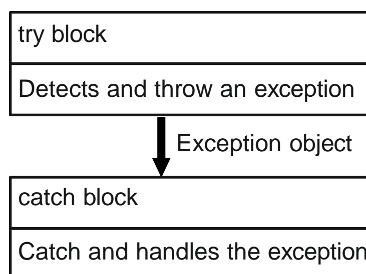


Figure 12.1 The block throwing exception The following program shows the use of try-catch blocks.

```
Program12.1 #include<iostream> using namespace std; int main()
{
int a,b;
cout<<"enter the values of a and b"; cin>>a;
cin>>b;  int x = a- b;
try
{
if(x!=0)
{
cout<<"result(a/x) = "<<a/x<<"\n";
```

```
}
else
{
throw(x);
}
}
catch(int i)
{
cout<<"exception caught : x = "<<x<<"\n";
}
cout<<"end"; return 0;
}
```

Output:

enter value of a and b

20 15

result(a/x)=4 end Second run

enter value of a and b

10 10

exception caught:x=0 end

The program detects and catches a division by zero problems. The output of first run shows a successful execution. When no exception is thrown, the catch statement is skipped and execution resumes with the first line after the catch. In the second run the denominator x become zero and therefore a division by zero situation occurs. This exception is thrown using the object x. Since the exception object is of integer type, the catch statement containing int type argument catches the exception and displays necessary message.

The exceptions are thrown by functions that are invoked from within the try block. The point at which the throw is executed is called throw point. Once an exception is thrown to catch block, control cannot return to the throw point.

The general format of code for this kind of relationship is shown below type function (arg list)

```
//function with exception
{ ………..
…………… throw(object);
//throw exception
```

………..

………..

}

………

…………. try

{ ……….

………. Invoke function here

………..

}

Catch (type arg)

//catches exception

{

…………..

………….

Handle exception here

………….

}

………..

It is to be noted here that the try block is immediately followed by the catch block irrespective of the location of the throw point.

The below program demonstrates how a try block invokes a function that generates an exception

Program 12.2

```
//Throw point outside the try block
#        include <iostream> using namespace std; void divide (int x,int y,int z)
{
cout<<"we are outside the function"; if ( ( x-y) != 0)
{ int r=z/(x-y); cout<<"result = "<<r;
}

else
{
throw(x-y);
}
```

```
}
int main()
{
try
{
cout<<"we are inside the try block"; divide(10,20,30); divide(10,10,20);
}
catch (int i)
{
cout<<"caught the exception";
}
return 0;
}
```

The output of the above program is We are outside the try block

We are inside the function Result =-3

We are inside the function Caught the exception

12.3 Throwing mechanism:-

When an exception is encountered it is thrown using the throw statement in the following form:

throw (exception); throw exception; throw;

The operand object exception may be of any type including constants. It is also possible to throw objects not intended for error handling. When an exception is thrown, it will be caught by the catch statement associated with the try block. In other words the control exits the try block and transferred to catch block after the try block. Throw point can be in the deep nested scope within the try block or in a deeply nested function call.

12.4 Catching mechanism:-

Code for handling exceptions is included in catch blocks. The catch block is like a function definition and is of form

```
Catch(type arg)
{  statements for managing exceptions
```

The type indicates the type of exception that catch block handles. The parameter arg is an optional parameter name. The catch statement catches an exception whose type matches with

the type of catch argument. When it is caught, the code in the catch block is executed. After executing the handler, the control goes to the statement immediately following in catch block. Due to mismatch, if an exception is not caught abnormal program termination wills occur. In other words catch block is simply skipped if the catch statement does not catch an exception.

12.4.1 Multiple Catch Statements:-

In some situations the program segment has more than one condition to throw an exception.In such case more than one catch blocks can be associated with a try block as shown below

```
try
{//try block
}
catch(type1 arg)
{
//catch block1
}
catch(type 2 arg)
{
//catch block 2
}
……………..
…………….
catch (type N arg)
{
//catch block N
}
```

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is executed. After executing the handler, the control goes to the first statement after the last catch block for that try. When no match is found, the program is terminated. If in some case the arguments of several catch statements match the type of an exception, then the first handler that matches the exception type is executed.

The below program shows the example of multiple catch statements.

Program12.3: MULTIPLE CATCH STATEMENTS

```
#include <iostream> using namespace std; void test (int x)
```

```
{
try
{
if (x==1) throw x;       //int else
if(x==0) throw 'x';      //char else
if (x== -1 ) throw 1.0;  //double cout<<"end of try- block \n";
}
catch(char c) //Catch 1
{
cout<<"Caught a character \n";
}
catch (int m) //Catch 2
{ cout <<"caught an integer\n";
}
catch (double d) //catch 3
{   cout<<"caught a double \n";
}
cout<<"end of try –catch system \n\n";
}
int main()
{
cout<<"Testing multiple catches \n"; cout<<"x== 1 \n"; test(1); cout<<"x== 0 \n";
test(0); cout<<"x == -1 \n"; test (-1); cout <<"x== 2 \n"; test (2);
return 0;
}
```

The program when executed first invokes the function test() with x=1 and throws x an int exception. This matches the type of parameter m in catch 2 and therefore catch2 handler is executed. Immediately after the execution , the function throws 'x', a character type exception and therefore the first handler is executed.Finally the handler catch3 is executed when a double type exception is thrown. Every time only the handler which catches the exception is executed and all other handlers are bypassed.

12.4.2 Catch All Exceptions:-

In some cases when all possible type of exceptions cannot be anticipated and may not be able to

design independent catch handlers to catch them, in such situations a single catch statement is forced to catch all exceptions instead of certain type alone.

This can be achieved by defining the catch statement using ellipses as follows catch(. . .)

```
{
//statement for processing all exceptions
}
```

The below program illustrate the functioning of catch(…) Program 12.4: CATCHING ALL EXCEPTIONS

```
#include <iostream> using namespace std; void test(int x)
{
try
{
if (x== 0) throw x;        //int
if ( x== -1)       throw 'x'; //char

if ( x== 1)        throw 1.0; //float
}
catch(. . .)
//catch all
{
cout<<"caught an exception \n";
}
}
int main()
{
cout<<"testing generic catch\n"; test(-1);
test(0);
test(1); return 0;}
```

We can use the catch(. . .) as a default statement along with other catch handlers so that it can catch all those exceptions that are not handled explicitly.

12.5 Rethrowing an Exception:-

A handler may decide to rethrow an exception caught without processing them.In such situations we can simply invoke throw without any argument like

throw;

This cause the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block. The following program shows how an exception is rethrown and caught.

Program 12.5: RETHROWING AN EXCEPTION

```
#include <iostream> using namespace std;
void divide(double x, double y)
{
cout<<"Inside Function \n"; try
{   if (y== 0.0)
throw y; //throwing double else
cout<<"division = "<< x/y<<"\n";
}

catch(double)   //Catch a double
{
cout<<"Caught double inside a function \n"; throw;     //rethrowing double
}
cout<<"end of function\n\n";
}
int main()
{
cout <<"inside main \n"; try
{ divide(10.5,2.0);
divide(20.0,0.0);
}
catch (double)
{ cout <<"caught double inside main \n";
}
cout <<"End of mai\n "; return 0;
}
```

When an exception is rethrown, it will not be caught by the same catch statement or any other catch in that group. It will be caught by the an appropriate catch in the outer try/catch sequence

for processing.

12.6 Specifying Exceptions:-

In some cases it may be possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a throw list clause to function definition. The general form of using an exception specification is:

```
Type function (arg-list) throw (type-list)
{
…………..

…………        function body

……….

}
```

The type list specifies the type of exceptions that may be thrown. Throwing any other type of exception will cause abnormal program termination. To prevent a function from throwing any exception, it can be done by making the type list empty like

throw(); //empty list

in the function header line.

The following program will show this

Program 12.6: TESTING THROW RESTRICTIONS

```
#include <iostream> using namespace std;
void test (int x) throw (int,double)
{
if (x== 0) throw 'x'; else
//char
if (x== 1) throw x;
//int else
if (x== -1) throw 1.0;  //double cout <<"End of function block \n ";
}
int main()
{
```

```
try
{
cout<<"testing throw restrictions\n"; cout<<"x== 0\n ";
test (0); cout<<"x==1 \n"; test(1); cout<<"x== -1 \n"; test(-1);
cout <<"x== 2 \n"; test(2);
}

catch( char c)
{
cout <<"caught a character \n";
}
catch(int m)
{
cout<<"caught an integer \n";
}
catch (double d)
{
cout<<"caught a double \n";
}
cout<<" end of try catch system \n \n"; return 0;
}
```

12.7 Summary

Exceptions are peculiar problems that a program may encounter at run time. ANSI C++ has built in language function for trapping the errors and controlling the exceptions. All C ++ compilers support this newly added facility. An exception is an object. It is send from the part of the program where an error occurs to the part of program which is going to control the error. C++ exception method provides three keywords, try, throw and catch. The keyword try is used at the starting of exception. The entire exception statement are enclosed in the curly braces. It is known as try block. The catch block receives the exception send by the throw block in the try block. Multiple catch blocks can be used in a program. It is also possible to define single or default catch () block from one or more exception of different type. In such situation a single catch block is used for catch exceptions thrown by the multiple throw statement. It is also possible to again pass the exception received to another exception handler

i.e an exception is thrown from catch() block and this is known as rethrowing the exception. The specified exception are used when we want to bind the function to throw only specified exceptions. Using a throw list condition can also do this.

12.8 Self Assessment Questions

1. What do you mean by exception handling?
2. Describe the role of keywords try, throw and catch in exception handling?
3. When should a program throw an exception?
4. What is an exception specification? When is it used?
5. When do we used multiple catch handlers?
6. Explain mechanism of exception handling.

Chapter 13

Introduction to Standard Template library

Objective:

At the end of this session students shall be learning:

- Introduction,
- The STL Programming Model,
- Understand Containers,
- Algorithms,
- Iterators,
- Function Objects,
- Understand Allocators,
- Understand Adaptors

Structure:

13.1. Introduction

13.2. Algorithms

13.3. C++: Containers in STL

13.4. C++: Iterators in STL

13.5. Use and Application of STL

13.6. Classification of Containers in STL

13.7. Using Container Library in STL

13.8. Standard Exceptions in C++

13.9. Dynamic Memory Allocation in C++

13.10. Summary.

13.11. Self-Assessment Questions

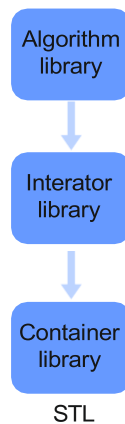
13.1. Introduction

In this chapter, we shall be discussing STL is an acronym for standard template library. It is a set of C++ template classes that provide generic classes and function that can be used to implement data structures and algorithms.

13.2. The C++ Standard Template Library (STL)

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms and iterators. It is a generalized library and so, its components are parameterized. STL is mainly composed:

1. Algorithms
2. Containers
3. Iterators



STL provides numerous containers and algorithms which are very useful in completive programming , for example you can very easily define a linked list in a single statement by using list container of container library in STL , saving your time and effort.

STL is a generic library , i.e a same container or algorithm can be operated on any data types , you don't have to define the same algorithm for different type of elements.

For example , sort algorithm will sort the elements in the given range irrespective of their data type , we don't have to implement different sort algorithm for different datatypes.

A working knowledge of template classes is a prerequisite for working with STL. STL has four components

- Algorithms
- Containers
- Functions
- Iterators

13.2 Algorithms

The header algorithm defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers.

1.Algorithm

- Sorting
- Searching
- Important STLAlgorithms
- Useful Array algorithms
- Partition Operations
- Numeric valarray class

STL provide number of algorithms that can be used of any container, irrespective of their type. Algorithms library contains built in functions that perform complex algorithms on the data structures. For example: one can reverse a range with reverse() function, sort a range with sort() function, search in a range with binary search() and so on. Algorithm library provides abstraction, i.e you don't necessarily need to know how the the algorithm works.

13.3 C++: Containers in STL

Container library in STL provide containers that are used to create data structures like arrays, linked list, trees etc. These container are generic, they can hold elements of any data types, for example: vector can be used for creating dynamic arrays of char, integer, float and other types.

13.4 C++: Iterators in STL

Iterators in STL are used to point to the containers. Iterators actually act as a bridge between containers and algorithms.

For example: sort() algorithm have two parameters, starting iterator and ending iterator, now sort() compare the elements pointed by each of these iterators and arrange them in sorted order, thus it does not matter what is the type of the container and same sort() can be used on different types of containers.

13.5 Use and Application of STL

STL being generic library provide containers and algorithms which can be used to store and manipulate different types of data thus it saves us from defining these data structures and algorithms from the scratch. Because of STL, now we do not have to define our sort function every time we make a new program or define same function twice for the different data types, instead we can just use the generic container and algorithms in STL. Such code, saves a lot of

time, code and effort during programming, thus STL is heavily used in the competitive programming, plus it is reliable and fast.

Containers Library in STL gives us the Containers, which in simplest words, can be described as the objects used to contain data or rather collection of object. Containers help us to implement and replicate simple and complex data structures very easily like arrays, list, trees, associative arrays and many more.

The containers are implemented as generic class templates, means that a container can be used to hold different kind of objects and they are dynamic in nature!

Following are some common containers:

·vector : replicates arrays

·queue : replicates queues

·stack : replicates stack

·priority queue : replicates heaps

·list : replicates linked list

·set : replicates trees

·map : associative arrays

13.6 Classification of Containers in STL

Containers are classified into four categories:

- **Sequence containers**: Used to implement data structures that are sequential in nature like arrays(array) and linked list(list).
- **Associative containers**: Used to implement sorted data structures such as map, set etc.
- **Unordered associative containers**: Used to implement unsorted data structures.
- **Containers adaptors**: Used to provide different interface to the sequence containers.

13.7 Using Container Library in STL

Following example provides an implementing linked list, first by using structures and then by list containers.

```
#include <iostream> struct node
{
int data;
struct node next;
}
```

```
int main ()
{
struct node list1 = NULL;
}
```

In the above program is only creating a list node, no insertion and deletion functions are defined, to do that, you will have to write more line of code.

Now let's see how using Container Library simplifies it. When we use list containers to implement linked list we just have to include the list header file and use list constructor to initialize the list.

```
#include <iostream> #include <list>
int main ()
{
list<int> list1;
}
```

We have a list, and not just that, the containers library also give all the different methods which can be used to perform different operations on list such as insertion, deletion, traversal etc. Thus you can see that it is incredibly easy to implement data structures by using Container library.

Containers or container classes store objects and data. There are in total seven standard "first-class" container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

- Sequence Containers: implement data structures which can be accessed in a sequential manner.
- vector
- list
- deque
- arrays
- forward_list
- Container Adaptors : provide a different interface for sequential containers.
- queue
- priority_queue
- stack
- Associative Containers: implement sorted data structures that can be quickly

searched (O(log n) complexity).

- set
- multiset
- map
- multimap

Functions in the STL include classes that overload the function call operator. Instances of such classes are called function objects or functions. Functions allow the working of the associated function to be customized with the help of parameters to be passed.

- ·Functions
- Iterators, as the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.
- Iterators Utility Library
- Defined under <utility header>
- ·pair

13.8 Standard Exceptions in C++

There are some standard exceptions in C++ under <exception> which we can use in our programs. They are arranged in a parent-child class hierarchy which is depicted below:

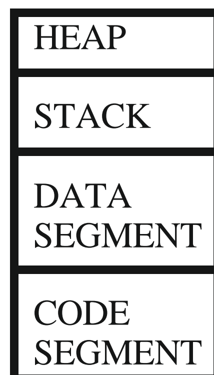
- ·std::exception - Parent class of all the standard C++ exceptions.
- ·logic_error - Exception happens in the internal logical of a program. o.domain_error - Exception due to use of invalid domain.
- o.invalid argument - Exception due to invalid argument.
- o.out_of_range - Exception due to out of range i.e. size requirement exceeds allocation.
- olength_error - Exception due to length error.
- ·runtime_error - Exception happens during runtime.
- o.range_error - Exception due to range errors in internal computations. o.overflow_error - Exception due to arithmetic overflow errors. o.underflow_error - Exception due to arithmetic underflow errors
- ·bad_alloc - Exception happens when memory allocation with new() fails.
- ·bad_cast - Exception happens when dynamic cast fails.
- ·bad_exception - Exception is specially designed to be listed in the dynamic-

exception-specifier.

- ·bad_typeid - Exception thrown by typeid.

13.9 Dynamic Memory Allocation in C++

Below is basic memory architecture used by any C++ program:



- Code Segment: Compiled program with executive instructions are kept in code segment. It is read only. In order to avoid over writing of stack and heap, code segment is kept below stack and heap.
- Data Segment: Global variables and static variables are kept in data segment. It is not read only.
- Stack: A stack is usually pre-allocated memory. The stack is a LIFO data structure. Each new variable is pushed onto the stack. Once variable goes out of scope, memory is freed. Once a stack variable is freed, that region of memory becomes available for other variables. The stack grows and shrinks as functions push and pop local variables. It stores local data, return addresses, arguments passed to functions and current status of memory.
- Heap: Memory is allocated during program execution. Memory is allocated using new operator and deallocating memory using delete operator.

13.10 Summary

In this chapter, we have learned the standard template library. It is a set of C++ template classes that provide generic classes and function that can be used to implement data structures and algorithms. In addition to this, we have understand Containers, Algorithms,

Iterators, Function Objects, understand Allocators, and understand Adaptors.

13.11 Self Assessment Questions

1. What is a stream? Describe briefly the features of I/O system supported by C++.
2. How is cout able to display various types of data without any special instructions?
3. Why it is necessary to include the file iostream in all our programs?
4. What is the role of iomanip file? What is the basic difference between manipulators and ios member functions in implementation? Give examples.
5. What are input and output streams?
6. What are the various classes available for file operations.
7. What is a file mode ?describe the various file mode options available.
8. Describes the various approaches by which we can detect the end of file condition.
9. What do you mean by command line arguments?

Chapter 14

Namespace

Objective:

At the end of this session students shall be learning:

- Introduction
- Defining a Namespace,
- The Standard Namespace,
- Nested Namespace,
- Unnamed Namespace,
- Namespace Alias

Structure:

14.1. Introduction

14.2. Creating a Namespace

14.3. Rules to create Namespaces

14.4. Using a Namespace in C++

14.5. Discontinuous Namespaces

14.6. Nested Namespaces

14.7. Summary.

14.8. Self Assessment Questions

14.1. Introduction

So far we have learned and used Namespace. Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifier defined in the namespace scope we must include the using directive, like

Using namespace std;

Here, std is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global scope. Using and namespace are the new keyword of C++.

Using namespace std, tells the compiler to use standard namespace. Namespace collects identifiers used for class, object and variables. Namespace can be used by two ways in a program, either by the use of using statement at the beginning, like we did in above mentioned program or by using name of namespace as prefix before the identifier with scope resolution (::) operator.

14.2. Creating a Namespace

Namespace is a container for identifiers. It puts the names of its members in a distinct space so that they don't conflict with the names in other namespaces or global namespace.

Creating a namespace is similar to creation of a class. namespace MySpace

```
{
// declarations
}
int main()
{        // main function
}
```

This will create a new namespace called MySpace, inside which we can put our member declar1ations.

14.3. Rules to create Namespaces

1. The namespace definition must be done at global scope, or nested inside another namespace.
2. Namespace definition doesn't terminates with a semicolon like in class definition.
3. You can use an alias name for your namespace name, for ease of use. Example for Alias:

```
namespace Study To night Dot Com
{
void study(); class Learn
{
// class defintion
};
}
// St is now alias for StudyTonightDotCom namespace St = StudyTonightDotCom;
```

4. You cannot create instance of namespace.

5. There can be unnamed namespaces too. Unnamed namespace is unique for each translation unit. They act exactly like named namespaces.

Example for Unnamed namespace:

```
namespace
{
class Head
{
// class defintion
};
// another class class Tail
{
// class defintion
};
int i,j,k;
}
int main()
{
// main function
}
```

6. A namespace definition can be continued and extended over multiple files, they are not redefined or over ridden.

For example, below is some header1.h header file, where we define a namespace: namespace MySpace

```
{
```

```
int x; void f();
}
```

We can then include the header1.h header file in some other header2.h header file and add more to an existing namespace:

```
#include "header1.h"; namespace MySpace
{
int y; void g();
}
```

14.4. Using a Namespace in C++

There are three ways to use a namespace in program,

1. Scope resolution operator (::)
2. The using directive
3. The using declaration

14.4.1. With Scope resolution operator (::)

Any name (identifier) declared in a namespace can be explicitly specified using the namespace's name and the scope resolution :: operator with the identifier.

```
namespace MySpace
{
class A
{
static int i; public:
void f();
};
// class name declaration class B;
//gobal function declaration void func();
}
// Initializing static class variable int MySpace::A::i=9;
class MySpace::B
{
int x; public:
int getdata()
{
```

```
cout << x;
}
// Constructor declaration B();
}

// Constructor definition MySpace::B::B()
{
x=0;
}
```

14.4.2. The using directive

using keyword allows you to import an entire namespace into your program with a global scope. It can be used to import a namespace into another namespace or any program.

Consider a header file Namespace1.h: namespace X

```
{
int x;
class Check
{
int i;
};
}
```

Including the above namespace header file in Namespace2.h file: #include "Namespace1.h";

```
namespace Y
{
using namespace X; Check obj;
int y;
}
```

We imported the namespace X into namespace Y, hence class Check will now be available in the namespace Y.

Hence we can write the following program in a separate file, let's say program1.cpp #include "Namespace2.h";

```
void test()
{
using Namespace Y;
```

// creating object of class Check Check obj2;

}

Hence, the using directive makes it a lot easier to use namespace, wherever you want.

14.4.3. The using declaration

When we use using directive, we import all the names in the namespace and they are available throughout the program, that is they have a global scope. But with using declaration, we import one specific name at a time which is available only inside the current scope.

Note here that, the name imported with using declaration can override the name imported with usingdirective

Consider a file Namespace.h: namespace X

```
{
void f()
{
cout << "f of X namespace\n";
}
void g()
{
cout << "g of X namespace\n";
}
```

Now let's create a new program file with name program2.cpp with below code:

```
#include "Namespace.h";
void h()
{
   using namespace X; // using directive
   using Y::f; // using declaration
   f();    // calls f() of Y namespace
   X::f(); // class f() of X namespace
}
f of Y namespace
f of X namespace
```

In using declaration, we never mention the argument list of a function while importing it, hence if a namespace has overloaded function, it will lead to ambiguity

```
#include "Namespace.h"; void h()
{
using namespace X; // using directive using Y::f; // using declaration
f();        // calls f() of Y namespace X::f(); // class f() of X namespace
}
#include "Namespace.h"; void h()
{
using namespace X; // using directive using Y::f; // using declaration
f();        // calls f() of Y namespace X::f(); // class f() of X namespace
}
}
// second name space namespace second_space{
void func(){
cout << "Inside second_space" << endl;
}
}
using namespace first_space; int main ()
{
// This calls function from first name space. func();
return 0;
}
```

If we compile and run above code, this would produce the following result: Inside first_space

The 'using' directive can also be used to refer to a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows:

```
using std::cout;
```

Subsequent code can refer to cout without prepending the namespace, but other items in the std namespace will still need to be explicit as follows:

```
#include <iostream> using std::cout;
int main ()
{
 cout << "std::endl is used with std!" << std::endl; return 0;}
```

If we compile and run above code, this would produce the following result: std::endl is used with std!

Names introduced in a using directive obey normal scope rules. The name is visible from the point of the using directive to the end of the scope in which the directive is found. Entities with the same name defined in an outer scope are
hidden.

14.5. Discontinuous Namespaces

A namespace can be defined in several parts and so a namespace is made up of the sum of its separately defined parts. The separate parts of a namespace can be spread over multiple files.

So, if one part of the namespace requires a name defined in another file, that name must still be declared. Writing a following namespace definition either defines a new namespace or adds new elements to an existing one:

```
namespace namespace_name {
// code declarations
}
```

14.6. Nested Namespaces

Namespaces can be nested where you can define one namespace inside another namespace as follows:

```
namespace namespace_name1 {
// code declarations
namespace namespace_name2 {
// code declarations
}
}
```

You can access members of nested namespace by using resolution operators as follows:

```
// to access members of namespace_name2
using namespace namespace_name1::namespace_name2;
// to access members of namespace: name1 using namespace namespace_name1;
```

In the above statements if you are using namespace_name1, then it will make elements of namespace_name2 available in the scope as follows:

```
#include <iostream> using namespace std;
```

```
// first name space namespace first_space{ void func(){
cout << "Inside first_space" << endl;
}
// second name space namespace second_space{
void func(){
cout << "Inside second_space" << endl;
}
}
}
using namespace first_space::second_space; int main ()
{
// This calls function from second name space. func();
return 0;
}
```

If we compile and run above code, this would produce the following result: Inside second_space

14.7. Summary.

In this chapter, we have introduction to the concept defining a Namespace, The Standard Namespace, Nested Namespace, Unnamed Namespace, Namespace Alias.

14.8. Self Assessment Questions

1. What is a namespace? Why it is necessary? Explain the advantages and disadvantages of namespaces.
2. What is a namespace? What are the rules for declaring namespace?
3. Write a note on :
   a. Nested namespace.
   b. scope resolution of namespace.

Chapter 15

New Style Caste and RTRI

Objective:

At the end of this session students shall be learning:

- Introduction,
- New-Style Casts,
- Static_cast,
- Dynamic_cast,
- Const_cast,Reinterpret_cast,
- A Simple Application of Run-Time Type ID,
- Typed can be applied to Templates classes

Structure:

15.1 Introduction

15.2 Type Casting

15.3 Const_cast

15.4 The New C++ Casting Operators

15.5 Reinterpret_cast

15.6 Summary

15.7 Self Assessment Questions

15.1. Introduction

Casts are used to convert the type of an object, expression, function argument, or return value to that of another type. Some conversions are performed automatically by the compiler without intervention by the programmer. These conversions are called implicit conversions. The standard C++ conversions and user-defined conversions are performed implicitly by the compiler where needed. Other conversions which must be explicitly specified by the programmer and are appropriately called explicit conversions.

Standard conversions are used for integral promotions (e.g., enum to int), integral conversions (e.g., int to unsigned int), floating point conversions (e.g., float to double), floating-integral conversions (e.g., int to float), arithmetic conversions (e.g., converting operands to the type of the widest operand before evaluation), pointer conversions (e.g.,

derived class pointer to base class pointer), reference conversions (e.g., derived class reference to base class reference), and pointer-to- member conversions (e.g., from pointer to member of a base class to pointer to member of a derived class).

Type conversion (often a result of type casting) refers to changing an entity of one data type, expression, function argument, or return value into another. This is done to take advantage of certain features of type hierarchies. For instance, values from a more limited set, such as integers, can be stored in a more compact format and later converted to a different format enabling operations not previously possible, such as division with several decimal places' worth of accuracy.

In the object-oriented programming paradigm, type conversion allows programs also to treat objects of one type as one of another. One must do it carefully as type casting can lead to loss of data.

Automatic type conversion (or standard conversion) happens whenever the compiler expects data of a particular type, but the data is given as a different type, leading to an automatic conversion by the compiler without an explicit indication by the programmer.

When an expression requires a given type that cannot be obtained through an implicit conversion or if more than one standard conversion creates an ambiguous situation, the programmer must explicitly specify the target type of the conversion. If the conversion is impossible it will result in an error or warning at compile time. Warnings may vary depending on the compiler used or compiler options. This type of conversion is useful and relied upon to perform integral promotions, integral conversions, floating point conversions, floating-integral conversions, arithmetic conversions, pointer conversions.

15.2. Type Casting:

Type Casting is used to convert the type of a variable, function, object, expression or return value to another type. It is the process of converting one type into another.

In other words converting an expression of a given type into another is called type casting. C++ supports following 4 types of casting operators:

1. const_cast
2. static_cast
3. dynamic_cast
4. reinterpret_cast

15.3. const_cast:

const_cast is used to cast away the consents' of variables. Following are some interesting facts about const_cast. A const_cast operator is used to add or remove a const or volatile modifier to or from a type. In general, it is dangerous to use the const_cast operator, because it allows a program to modify a variable that was declared const, and thus was not supposed to be modifiable.

const_cast can be used to change non-const class members inside a const member function. Consider the following code snippet. Inside const member function fun(), 'this' is treated by the compiler as 'const student const this', i.e. 'this' is a constant pointer to a constant object, thus compiler doesn't allow to change the data members through 'this' pointer. const_cast changes the type of 'this' pointer to 'student const this'.

filter_none edit play_arrow brightness_

```
#include <iostream> using namespace std; class student
{
private:
int roll; public:
// constructor student(int r):roll(r) { }
// A const function that changes roll with the help of const_cast void fun() const
{
 ( const_cast <student> (this) )->roll = 5;
}
int getRoll()  { return roll; }
};
int main(void)
{
student s(3);
cout << "Old roll number: " << s.getRoll() << endl; s.fun();
cout << "New roll number: " << s.getRoll() << endl; return 0;
}
```

Output:

Old roll number: 3 New roll number: 5

const_cast can be used to pass const data to a function that doesn't receive const. For example, in the following program fun() receives a normal pointer, but a pointer to a const can be passed with the help of const_cast.

filter_none edit play_arrow brightness_4

```
#include <iostream> using namespace std; int fun(int ptr)
{
return (ptr + 10);
}
int main(void)
{
const int val = 10; const int ptr = &val;
int ptr1 = const_cast <int >(ptr); cout << fun(ptr1);
return 0;
}
```

Output:

20

3) It is undefined behavior to modify a value which is initially declared as const. Consider the following program. The output of the program is undefined. The variable 'val' is a const variable and the call 'fun(ptr1)' tries to modify 'val' using const_cast.

```
#include <iostream> using namespace std; int fun(int ptr)
{
ptr = ptr + 10; return (ptr);
}
int main(void)
{

const int val = 10; const int ptr = &val;
int ptr1 = const_cast <int >(ptr); fun(ptr1);
cout << val; return 0;
}
```

Output:

Undefined Behavior

It it fine to modify a value which is not initially declared as const. For example, in the above program, if we remove const from declaration of val, the program will produce 20 as output.

```
#include <iostream> using namespace std;
int fun(int ptr)
{
```

```
ptr = ptr + 10; return (ptr);
}
int main(void)
{
int val = 10;
const int ptr = &val;
int ptr1 = const_cast <int >(ptr); fun(ptr1);
cout << val; return 0;
}
```

4) const_cast is considered safer than simple type casting. It'safer in the sense that the casting won't happen if the type of cast is not same as original object. For example, the following program fails in compilation because 'int ' is being typecasted to 'char'

```
#include <iostream> using namespace std; int main(void)
{
int a1 = 40;
const int b1 = &a1;
char c1 = const_cast <char > (b1); // compiler error
c1 = 'A';
return 0;
}
```

output:

```
prog.cpp: In function 'int main()':
prog.cpp:8: error: invalid const_cast from type 'const int' to type 'char'
```

5) const_cast can also be used to cast away volatile attribute. For example, in the following program, the typeid of b1 is PVKi (pointer to a volatile and constant integer) and typeid of c1 is Pi (Pointer to integer)

```
#include <iostream> #include <typeinfo> using namespace std;
int main(void)
{
int a1 = 40;
const volatile int b1 = &a1;
cout << "typeid of b1 " << typeid(b1).name() << '\n'; int c1 = const_cast <int > (b1);
```

```
cout << "typeid of c1 " << typeid(c1).name() << '\n'; return 0;
}
```

Output:

typeid of b1 PVKi typeid of c1 Pi

Exercise

Predict the output of following programs. If there are compilation errors, then fix them.

Question 1

```
#include <iostream> using namespace std;
class student
{
private:
const int roll; public:
// constructor student(int r):roll(r) { }
// A const function that changes roll with the help of const_cast void fun() const
{
( const_cast <student> (this) )->roll = 5;
}
int getRoll() { return roll; }
};
int main(void)
{

student s(3);
cout << "Old roll number: " << s.getRoll() << endl; s.fun();
cout << "New roll number: " << s.getRoll() << endl;
return 0;
}
```

Question 2

```
#include <iostream> using namespace std;
class student
{
```

```
private:
    const int roll; public:
// constructor student(int r):roll(r) { }
  // A const function that changes roll with the help of const_cast void fun() const
{



( const_cast <student> (this) )->roll = 5;
}
int getRoll()  { return roll; }
};
int main(void)
{
student s(3);
cout << "Old roll number: " << s.getRoll() << endl; s.fun();
cout << "New roll number: " << s.getRoll() << endl;
return 0;
}
```

You can provide a user-defined conversion from a class X to a class Y by providing a constructor for Y that takes an X as an argument:

Y(const X& x)

or by providing a class Y with a conversion operator: operator X()

When a type is needed for an expression that cannot be obtained through an implicit conversion or when more than one standard conversion creates an ambiguous situation, the programmer must explicitly specify the target type of the conversion.

In C, an expression, expr, of type S can be cast to another type T in one of the following ways.

By using an explicit cast:

(T) expr

or by using a functional form:

T(expr)

We will refer to either of these constructs as the old C-style casts.

15.4. The New C++ Casting Operators:

The new C++ casting operators are intended to provide a solution to the shortcomings of the old C-style casts by providing: 1. Improved syntax. Casts have a clear, concise, although somewhat cumbersome syntax. This makes casts easier to understand, find, and maintain. 2. Improved semantics. The intended meaning of a cast is no longer ambiguous. Knowing what the programmer intended the cast to do makes it possible for compilers to detect improper casting operations. 3. Type-safe conversions. Allow some casts to be performed safely at run-time. This will enable programmers to check whether a particular cast is successful or not.

15.5. The reinterpret_cast Operator

The reinterpret_cast operator takes the form reinterpret_cast<T> (expr) and is used to perform conversions between two unrelated types. The result of the conversion is usually implementation dependent and, therefore, not likely to be portable. You should use this type of cast only when absolutely necessary. A reinterpret_cast can also be used to convert a pointer to an integral type. If the integral type is then converted back to the same pointer type, the result will be the same value as the original pointer.

15.6. Summary

The introduction of the new C++ cast operators heralds a significant advancement in programming practices, facilitating the development of programs that are not only easier to maintain and comprehend but also ensure safer conversions. However, it's imperative to approach casting operations with caution and discernment. When transitioning from old C-style casts to utilize these modern casting operators, it's essential to introspect and question the necessity of each cast. There may arise instances where the use of a cast is not truly warranted. It could indicate a potential misuse of a class hierarchy or an opportunity to achieve the same outcome through the implementation of virtual functions. Therefore, while leveraging C++ cast operators, exercising prudence and considering alternative design patterns can lead to more robust and maintainable codebases.

15.7 Self Assessment Questions

1. What is the purpose of using new-style casts in C++?
2. How does the static_cast differ from other new-style casts in C++?
3. Can you explain the syntax and usage of static_cast in C++?
4. When would you use a static_cast in a C++ program? Provide examples.
5. What are the advantages of using static_cast compared to C-style casts?
6. What are the limitations or risks associated with using static_cast in C++?
7. How does dynamic_cast differ from static_cast in C++?
8. When would you use a dynamic_cast in a C++ program? Provide examples.
9. What are the scenarios where dynamic_cast is particularly useful?
10. What are the key differences between static_cast and dynamic_cast in terms of performance and behavior?