SPPU New Syllabus

A Book Of

Object Oriented Programming and C++

For B.C.A. (Science) : Semester - IV

[Course Code BCA (Science) 241]

CBCS Pattern

As Per New Syllabus, Effective from June 2020

Dr. Ms. Manisha Bharambe
M.Sc. (Comp. Sci), M.Phil, Ph.D (Comp. Sci)
Vice Principal and Associate Professor
Computer Science Department
Abasaheb Garware College, Pune - 4

Price 220.00

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N5551

Syllabus ...

1. **Introduction to C++** **[06 Hrs.]**
   - How C++ Evolved from C?
   - Features of C++
   - Procedure-Oriented Programming Vs Object-Oriented Programming
   - The Basic Anatomy of a C++ Program, Simple "Hello World" Program
   - Compiling, Linking and Running a C++ Program.
2. **Object Oriented Programming Concepts** **[08 Hrs.]**
   - Classes and Objects
   - Abstraction
   - Inheritance
   - Polymorphism
   - Data Binding
   - Encapsulation
   - Message Passing
3. **Programming in C++** **[08 Hrs.]**
   - Data Types
   - New Operators and Keywords
   - Type Casting in C++
   - reference Variables,
   - Classes and Access Specifiers
   - Defining Data Members and Member Functions
   - Arrays and Array of Objects
   - Usage of Namespace, Managing Console I/O, Usage of Manipulators
4. **Constructors, Destructors, Function and Operator overloading** **[14 Hrs.]**
   - Static Members Call by Reference, Return by Reference
   - Inline Function
   - Friend Function
   - Function Overloading
   - Constructer and Destructor and their types
   - Overloading Unary and Binary Operators (with Member Function and with Friend Function)
   - Usage of this Pointer
   - Overloading using Friend Functions Overloading "<<>>" Operator Type Conversion, Function Overloading

[14 Hrs.]

5. **Inheritance**
   - Introduction
   - Types of Inheritance
   - Base Class and Derived Class Examples
   - Virtual Base Class
   - Abstract Class
   - Polymorphism
   - Virtual Functions and Pure Virtual Functions, Overriding

[10 Hrs.]

6. **Files and Exception Handling**
   - Classes for File Stream Operations
   - Opening and Closing a File
   - Random Access Files
   - Introduction to Exception Handling Mechanism

■■■

Contents ...

1. **Introduction to C++** — **1.1 – 1.18**

2. **Object Oriented Programming Concepts** — **2.1 – 2.12**

3. **Programming in C++** — **3.1 – 3.38**

4. **Constructors, Destructors, Function and Operator Overloading** — **4.1 – 4.58**

5. **Inheritance** — **5.1 – 5.48**

6. **Files and Exception Handling** — **6.1 – 6.26**

■■■

1... Introduction to C++

Objectives...

- To learn How C++ evolved from C?
- To understand Features of C++.
- To study Difference between Procedure oriented programming and Object oriented programming.
- To study the basic anatomy of a C++ program, Simple "Hello World" program.
- To study about compiling, linking and running a C++ program.

1.1 INTRODUCTION

- C++ is an extension of C language, developed by Bjarne Stroustrup at AT&T's Bell Labs in the early 1980's.
- C++ is a general purpose programming language which has been derived from C programming language.
- C++ could be considered a superset of C language with extensions and improvements with object oriented features included in it. C++ runs on a variety of platforms, such as Windows, Mac OS, UNIX, LINUX etc.
- C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features.
- C++ is popular languages because of following reasons:
  1. C++ is ideally suited for development of reusable software.
  2. C++ is highly flexible language with versatility.

1.2 HOW C++ EVOLVED FROM C?

- C++ is an Object Oriented Programming (OOP) language developed by Bjarne Stroustrup. He is a master of SIMULA 67 and C, wanted to combine the features of both the languages and he developed a powerful language that supports object oriented programming with features of C.
- Various features were derived from SIMULA 67 and ALGOL 68. B. Stroustrup called the new language 'C with Classes'. However, in 1983, the name was changed to C++.

- Fig. 1.1 evolution of C++.

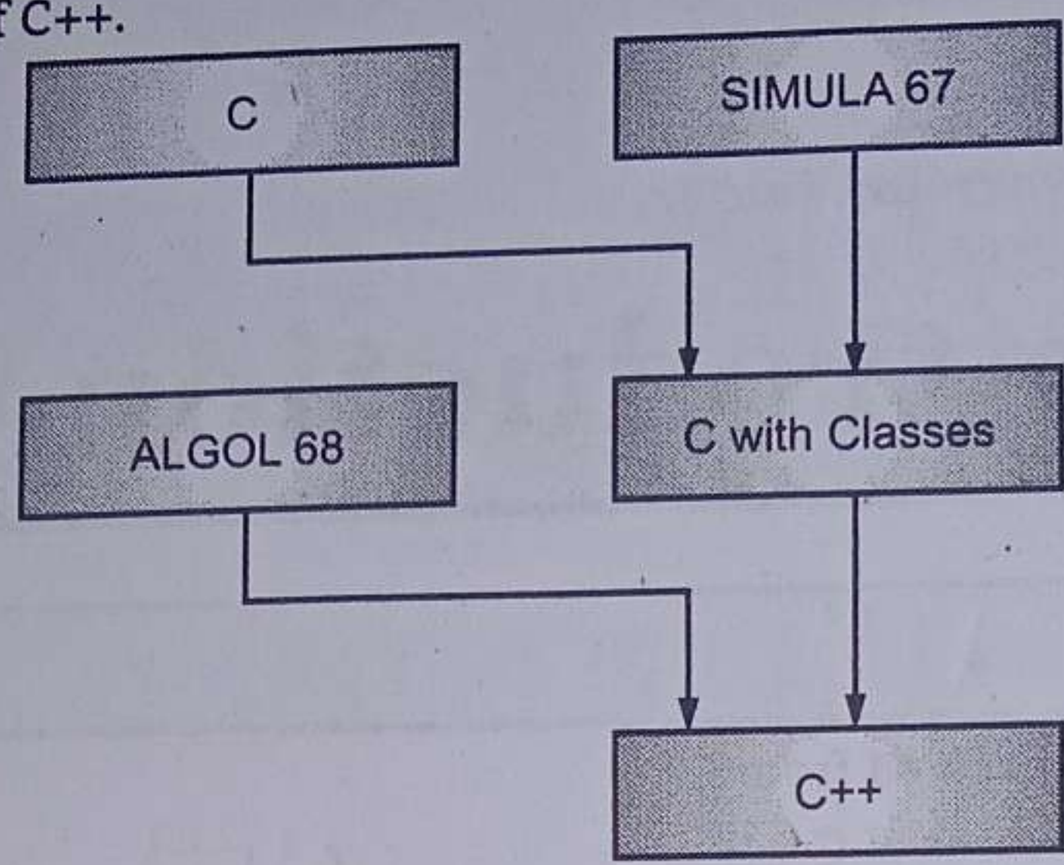


Fig. 1.1: Evolution of C++

- The thought of C++ came from the 'C' increment operator ++. Rick Mascitti coined the term C++ in 1983. Therefore, C++ is an extension of 'C' and a superset of 'C'. All the concepts of C are applicable to C++ also.
- For developing complicated applications, object oriented language such as C++ is the most convenient and easy.

1.3 FEATURES OF C++

- C++ language provides a lot of features that are given in Fig. 1.2.

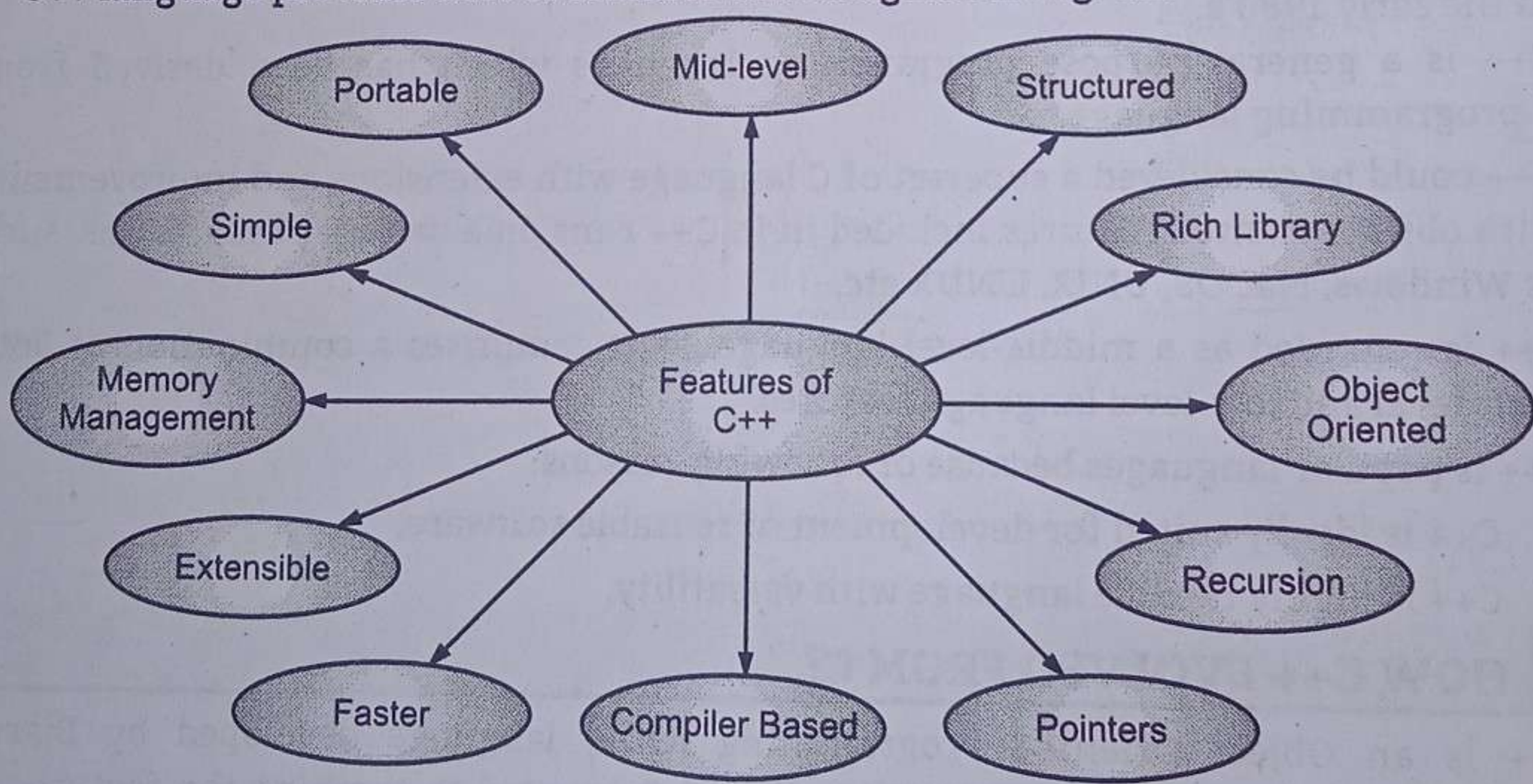


Fig. 1.2: Features of C++

- **Various features of C++ are explained below:**
  1. **Simple:** C++ is a simple language because it provides structured programming approach, rich set of library functions, data types etc.
  2. **Machine Independent or Portable:** C++ programs can be executed in different computer machines with little bit or no change.

3. **Middle-level Programming Language:** C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features.
4. **Structured Programming Language:** C++ is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.
5. **Rich Library:** C++ provides a lot of inbuilt functions that makes the development fast.
6. **Memory Management:** C++ supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory at any time by calling the free() function.
7. **Speed:** The compilation and execution time of C++ language is fast.
8. **Pointer:** C++ provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array etc.
9. **Recursion:** In C++, we can call the function within the function. It provides code reusability for every function.
10. **Extensible:** C++ language is extensible because it can easily adopt new features.
11. **Object Oriented:** C++ is object oriented programming language. OOPs make development and maintenance easier.
12. **Compiler Based:** C++ is a compiler based programming language, it means without compilation no C++ program can be executed. First we need to compile the program using compiler and then we can execute the program.

1.4 PROCEDURE ORIENTED PROGRAMMING VS OBJECT ORIENTED PROGRAMMING

1.4.1 Procedure Oriented Programming

- In this approach the problem is viewed as a sequence of things like reading, printing, calculating and so on.
- C, COBOL, FORTRAN and PASCAL are procedural oriented programming languages.
- Using programming procedure oriented approach the programmer views a problem as a sequence of things to do.
- The programmer organizes the related data items and writes the necessary functions to manipulate the data and the process, complete the sequence of tasks that solve the problem.
- POP (Procedure Oriented Programming) focus is on the functions.
- A program in a procedural language is a list of instructions. When program becomes larger, it is broken down into smaller units, called **functions** or **procedures**.
- Each function has a clearly defined purpose and a clearly defined interface to the other functions in the program. A number of functions can be grouped together into a larger entity called a **module**.

- Fig. 1.3 shows typical structure of procedure oriented programs.

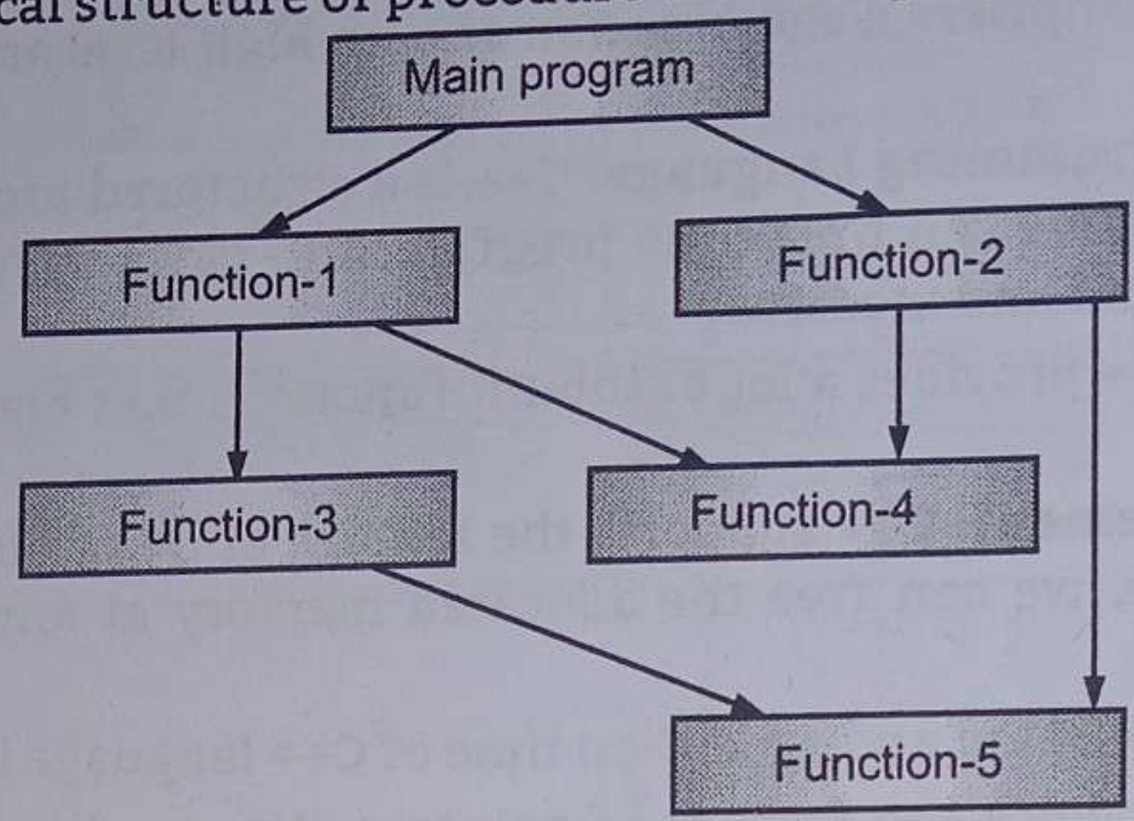


Fig. 1.3: Typical program structure of procedural oriented programming language

- A function is also called as procedure. A procedure call is used to invoke the procedure. After the sequence is processed, flow of control proceeds right after the position where the call was made, as shown in Fig. 1.4.
- But there are problems with procedure oriented programming (i.e. structured programming), as programs become larger and complex, they become difficult to handle. There are two related reasons, first, functions have unrestricted access to global data and second, unrelated functions and data provide a poor model of the real world.

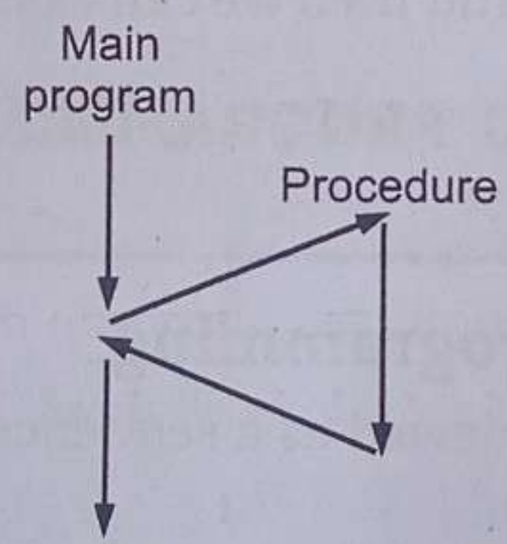


Fig. 1.4: Flow of control in procedural programming

- Let us see, what we mean by unrestricted access to global data. When we concentrate on the development of functions, very little attention is given to the data that are being used by these functions.
- There are two kinds of data: Local data, and Global data.
  1. **Local Data:** It has restricted access in one function only. It is hidden inside a function and can be used only within that function and it is safe from modifications by other functions.
  2. **Global data:** When more than one function want to access the same data, then the Global data is used. Global data can be accessed by any function in the program.
- In a large program, it is very difficult to identify what data is used by which function.

- Also it makes the program difficult to modify. When some global data is changed, it may be necessary to modify all the functions that access this data. The scenario is depicted in the Fig. 1.5.

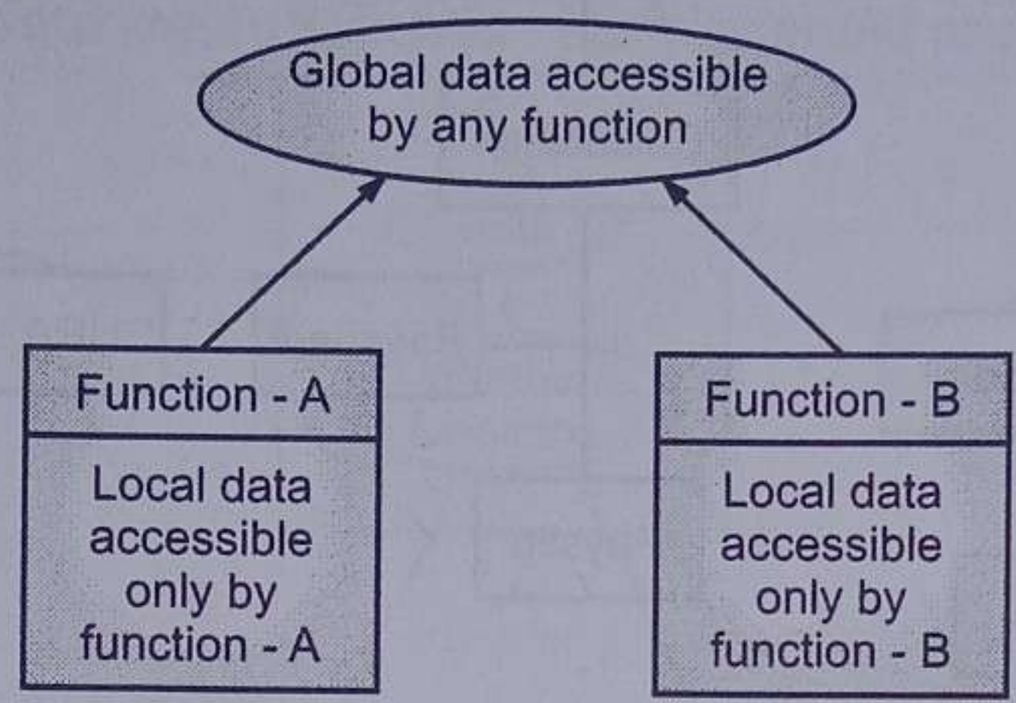


Fig. 1.5: Data access in procedural programming languages

- There is one more problem with procedural programming that it does a poor modeling of the things in the real world because of its arrangement of separate functions and data.
- Real world objects have both attributes and behaviour. Attributes like weight, height, name etc. and behaviour like eat, do, run, take, stop etc. So neither data nor functions model real-world objects effectively.
- **Characteristics of Procedural Oriented Programming:**
  1. OOP's focus is on the functions.
  2. Procedural programming languages uses top-down programming approach in program design, i.e. the main task is divided into subtasks and functions are written to perform each subtask. Then what data is needed is decided, i.e. emphasis is on algorithms instead of data.
  3. Different functions share the data through global variables.
  4. Data moves openly around the system from function to function because of global data are shared.
  5. The major trace is given on the algorithms (doing things).
  6. Functions transform data from one form to another form.
  7. Large programs are divided into smaller programs known as functions, (modules).
- For example, in Fig. 1.6 payroll system is explained. Payroll menu program is the main program. Emp_detail, other detail are subprogramming, Add View, Delete, Modify, HRA, DA, CLA are different functions.

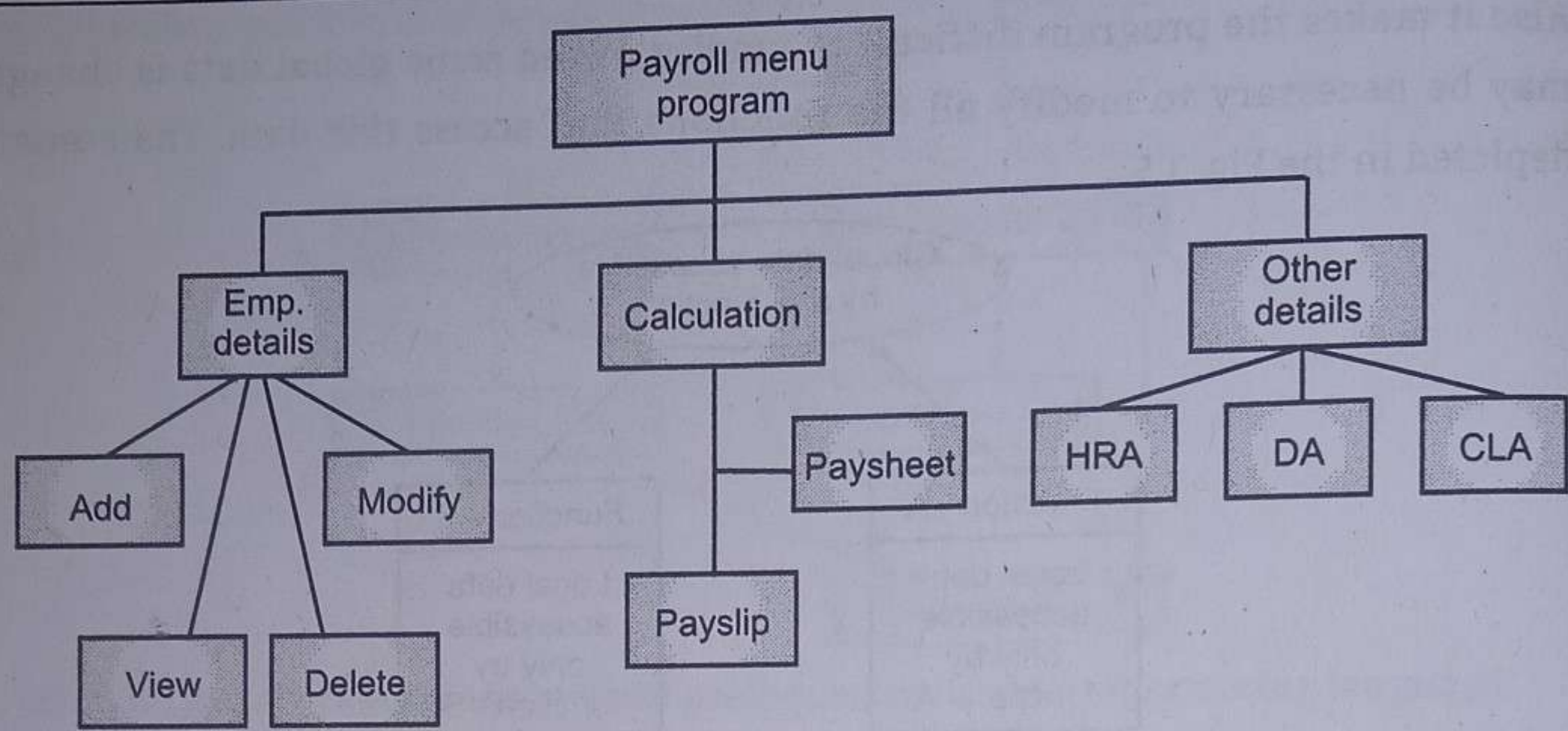


Fig. 1.6: Procedure oriented program structure for payroll

- The field or data item emp_id, emp_name, designation can be declared as global and can be shared by different functions. These functions may have local data also. This relationship is shown in Fig. 1.7.

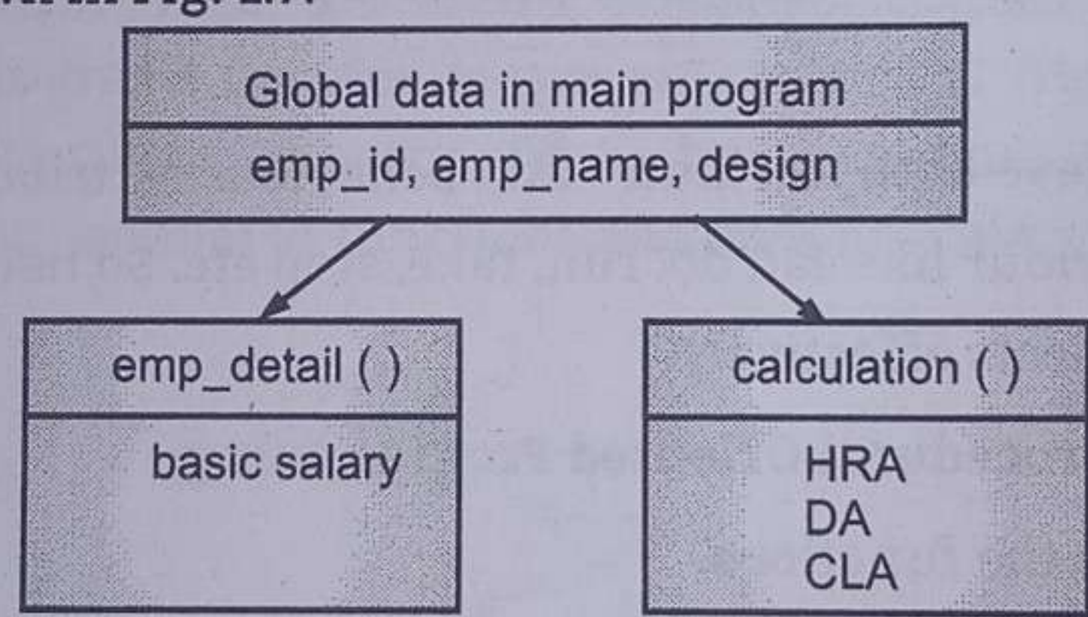


Fig. 1.7: Relationship of data and function in structured programming

- **Drawbacks of Procedure Oriented Programming:**
  1. We can modify data of any function from any other function.
  2. Data can be accessed from any part of a program i.e. all data is visible in the program.

1.4.2 Object Oriented Programming

- Object oriented programming is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.
- In OOP approach, the main focus is on the data that is to be used rather than the function.
- Once the data, which is to be used, is decided then the different functions that will operate on this data are defined.
- Many drawbacks of procedural languages are eliminated by object oriented languages. The object oriented languages combine's data and functions that operate on that data, into a single unit called as **object**. This helps to achieve data encapsulation and data hiding.

- An object encapsulates the data and its operations, thus hides the data from remaining program. Member functions allow other part of the program to access the data indirectly, direct access is restricted.
- The organisation of data and function in object-oriented program is shown in Fig. 1.8.

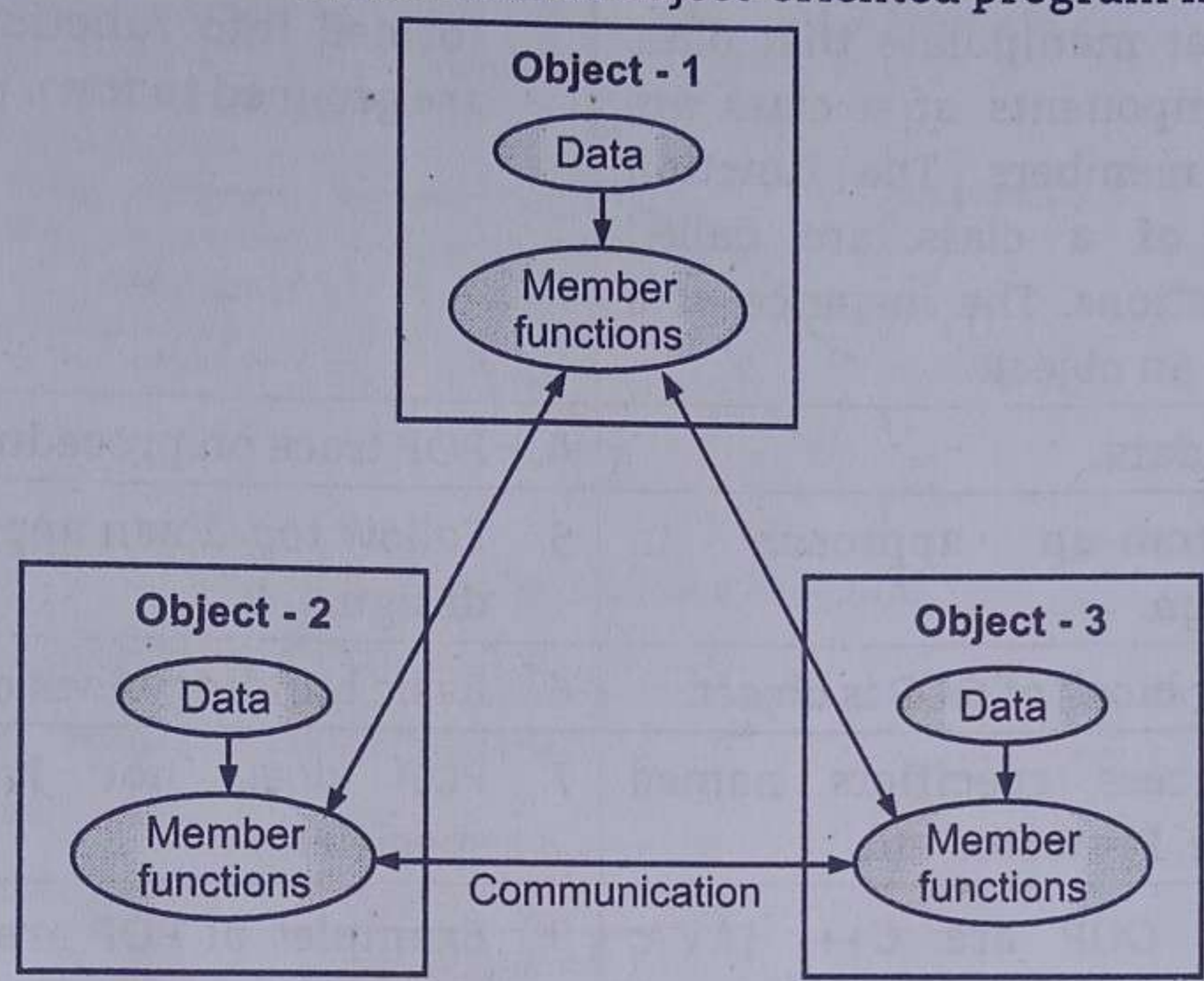


Fig. 1.8: Organization of an OOP program

- **Characteristics of Object Oriented Programming:**
  1. It uses bottom-up programming approach for design.
  2. Object oriented programming emphasis is on data rather than procedure.
  3. Data and its operations are bound together into objects.
  4. Data is hidden from the remaining part of the program.
  5. Objects can communicate with each other through message passing.
  6. Data can be organized in a hierarchical structure using inheritance.
  7. New data and functions can be easily added wherever necessary.
  8. Data structures are designed such that they characterize the objects.

1.4.3 Object Oriented Programming Vs Procedure Oriented Programming

[S-18]

Table 1.1: Object Oriented Programming Vs Procedural Oriented Programming

Object Oriented Programming (OOP)	Procedure Oriented Programming (POP)
1. Object oriented programming language is object oriented.	1. Procedural programming languages tend to be action oriented.
2. In OOP, the unit of programming is the class.	2. In procedural programming, unit of programming is the function.

contd. ...

3. OOP programmers concentrate on creating their own user-defined types called classes and components. Each class contains data as well as the set of functions that manipulate that data. The data components of a class are called data members. The function components of a class are called member functions. The instance of a class is called an object.	3. In procedural oriented programming programmers concentrate on writing functions. Groups of actions that perform some common task are formed into functions and functions are grouped to form programs.
4. OOP trace on data.	4. POP trace on procedures.
5. Follow bottom-up approach in program design.	5. Follow top-down approach in program design.
6. Basic building block of OOP is object.	6. Basic building block of POP is function.
7. OOP has access specifiers named Public, Private, Protected etc.	7. POP does not have any access specifiers.
8. Examples of OOP are C++, JAVA, VB.NET, C#.NET.	8. Examples of POP are C, VB, FORTRAN, Pascal.
9. In OOP, importance is given to the data rather than procedures or functions because it works as a real world.	9. In POP, importance is not given to data but to functions as well as sequence of actions to be done.
10. OOP provide Data Hiding so provides more security.	10. POP does not have any proper way for hiding data so it is less secure.
11. In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.	11. In POP, Overloading is not possible.
12. In OOP, objects can move and communicate with each other through member functions.	12. In POP, data can move freely from function to function in the system.

1.5 THE BASIC ANATOMY OF A C++ PROGRAM

- A computer program is a sequence of instructions that tell the computer what to do.
- A typical C++ program would contain four sections:
  1. Include file (Header file).
  2. Class declaration.
  3. Member function definition.
  4. Main function.

- Fig. 1.9 shows structure of a C++ program.

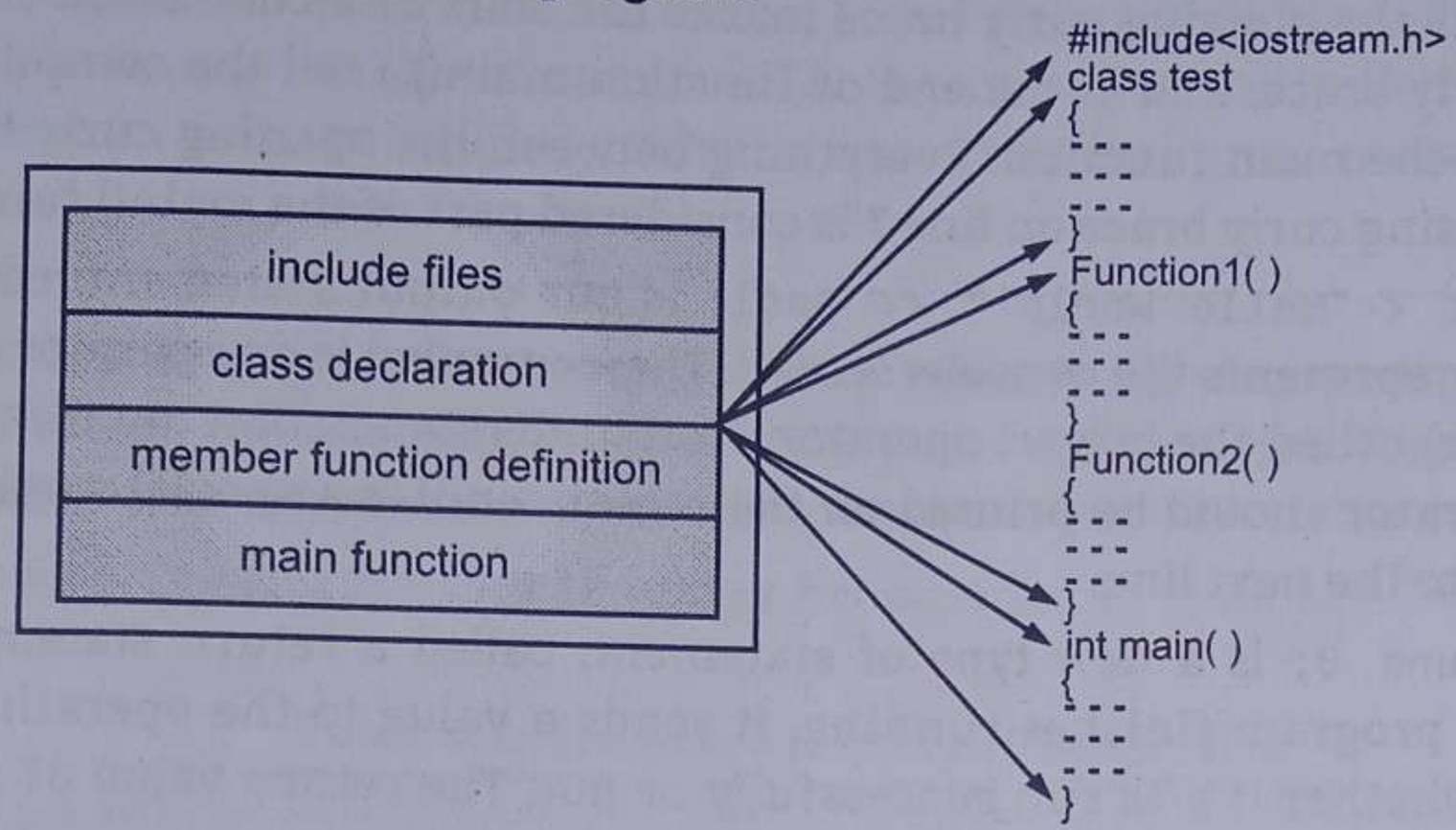


Fig. 1.9: Structure of C++ Program

- C++ program is a collection of functions.

Program 1.1: Simple "Hello World" program.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     cout << "Hello World!" << endl;
6.     return 0;
7. }
```

Output:

```
Hello World!
```

- Line 1 `#include<iostream>` is a special type of statement called a preprocessor directive and it start from #. Preprocessor directives tell the compiler to perform a special task. In this case, we are telling the compiler that we would like to use the iostream library. The iostream library contains code that tells the compiler what cout and endl do. In other words, we need to include the iostream library in order to write to the screen.
- Line 2 `using namespace std;` defines a scope for the identifiers that are used in the program. As you learned in the explanation for line 1, cout and endl live inside the iostream library. However, within iostream, they live inside a special compartment named std (short for standard). This using statement tells the compiler to look inside a compartment named std if it cannot find cout or endl defined anywhere else. In other words, this statement is also necessary so the compiler can find cout and endl, which we use on line 5.
- Line 3 `int main()` declares the main() function which is the point from where all C++ program begin their execution. Every program must have a main() function.

- Lines 4 ({ is the opening curly brace marks the start of a code block) and 7 (} is the closing curly brace, marks the end of function main()) tell the compiler which lines are part of the main function. Everything between the opening curly brace on line 4 and the closing curly brace on line 7 is considered part of the main() function.
- Line 5 `cout <<"Hello World!" << endl;` is our output statement. cout is a special object that represents the console/screen. The << symbol is an operator (much like + is an operator) called the output operator. cout understands that anything sent to it via the << operator should be printed on the screen. endl is a special symbol that moves the cursor to the next line.
- Line 6 `return 0;` is a new type of statement, called a return statement. When an executable program finishes running, it sends a value to the operating system that indicates whether it was run successfully or not. The return value of main () is used for this purpose. This particular return statement returns the value of 0 to the operating system, which means "everything went okay!".

1. **Comments:**

- Comments are parts of the source code disregarded by the compiler. They simply do nothing.
- Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.
- C++ supports two ways to insert comments:
  (i) // line comment, and
  (ii) /* */ block comment.
- The first of them, known as line comment, discards everything from where the pair of slash signs (//) is found up to the end of that same line.
- The second one, known as block comment, discards everything between the /* characters and the first appearance of the */ characters, with the possibility of including more than one line.

Program 1.2: Program for comments.

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World!" << endl;          //Prints Hello World!
    cout << "Wel Come to C++ !" << endl;     //Prints Welcome to C++!
    return 0;
}
```

Output:

```
Hello World!
Welcome to C++ !
Press any key to continue . . .
```

- If you include comments within the source code of your programs without using the comment characters combinations //, /* */, the compiler will take them as if they were C++ expressions, most likely causing one or several error messages when you compile it.

2. **Header Files:**

- Each standard library has a corresponding header file containing the function prototypes for all the functions in that library and definitions of various data types and constants needed by those functions.
- Some common C++ header files that may be included in standard C++ program, are given below:

Table 1.2: Standard Library Header Files

Standard library header files	Explanation
<assert.h>	Contains macros and information for adding diagnostics that aid program debugging.
<ctype.h>	Contains function prototypes for functions that test characters for certain properties and that can be used to convert lowercase letters to uppercase letters.
<float.h>	Contains the floating point size limits of one program.
<limits.h>	Contains integral size limits of the system.
<math.h>	Contains function prototype for math library function.
<stdio.h>	Contains function prototype for the standard input/output library functions and information used by them.
<stdlib.h>	Contains function prototype for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions.
<string.h>	Contains function prototype for C-style string processing function.
<time.h>	Contains function prototype for manipulating time and date.
<iostream.h>	Contains function prototype for standard input and output functions.
<iomanip.h>	Contains function prototype for the stream manipulators that enable formatting of streams of data.
<fstream.h>	Contains function prototype for functions that perform input from files on disk and output to files on disk.

- The programmer can create custom header files. Programmer-defined files should end in .h, which can be included by using the # include preprocessor directive.

- For example, the header file square.h can be included in our program by the directive,

```
# include "square.h"
```

at the top of the program.

3. Streams in C++:

- C++ support a set of functions for performing input and output operations. The new features of C++ is called **streams** which is used to handle I/O operations.
- C++ streams are of two types:
  (i) Input stream, and
  (ii) Output stream.
- Stream refers to the flow of data from a particular source to a specified destination.
- There are classes representing each stream **istream** is the class representing the input stream (istream) and **ostream** is the class representing the output stream.
- To achieve the console input/output operations, we use the objects of these stream classes.

(i) Input Stream:

- The input stream is handling all input operations or reading operations with input devices like keyboard, disk etc. **"cin"** is the predefined object of the istream class. It is used for all console input operations.
- Operator >>, (or the right shift operator in C) is called the **extraction** operator, it gets the value from the stream object **cin** on its left and places it in the variable on its right.

Syntax:

```
cin>>variable;
```

- The use of cin is shown in Fig. 1.10.

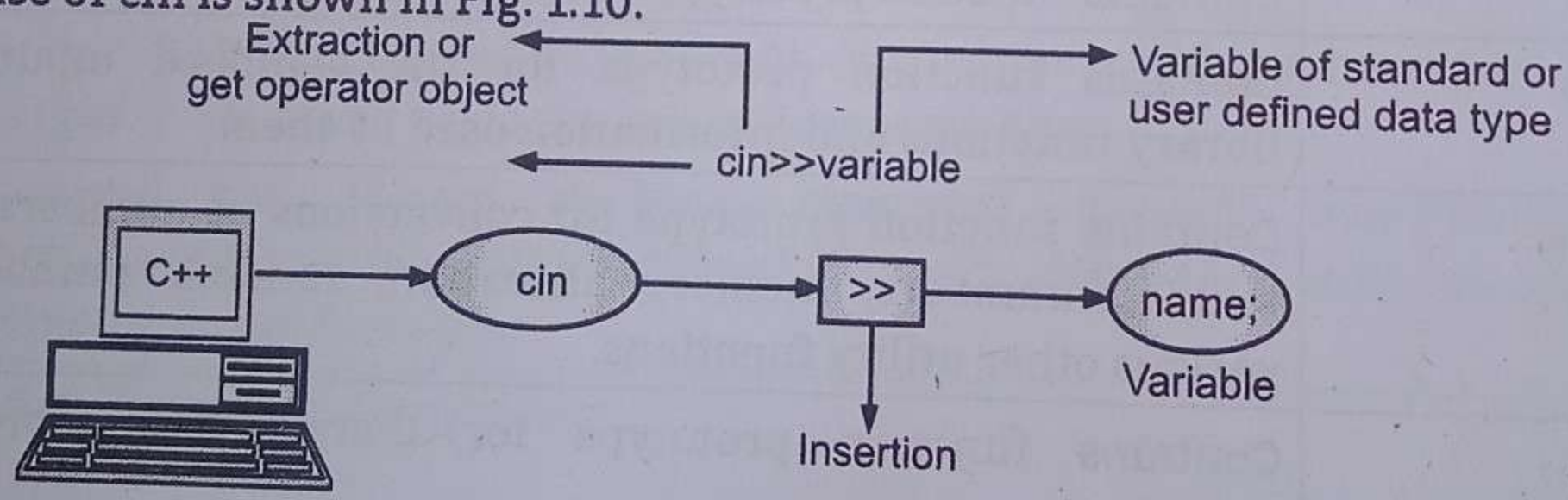


Fig. 1.10: Input with cin Operator

Examples:

1.
```
int a;
cin>>a;
```
2.
```
float salary;
cin>>salary;
```
3.
```
double area;
cin>>area;
```
4.
```
char name[15];
cin>>name;
```

- We can input more than one item using cin object. Such input operations are called cascaded input operations.
- cin will read all the items from left to right.

Syntax for more than one variable as:

```
cin>>var1>>var2>>var3........ >>varn;
```

Examples:

```
1. cin>>x>>y;          //x read first then y
2. cin>>name>>address;
3. cin>>roll>>name>>marks;
```

(ii) Output Stream:

- The output stream handles the write operations on output devices like screen, disk etc. **cout** is the predefined object of the **stream** class and is used for all console output operations.
- Operator **<<**, (or the left shift operator in C) is called **insertion** operator. It directs the contents of the variable on its right to the object (cout) on its left.

Syntax:

```
cout<<variable;
```

- The use of cout is shown in Fig. 1.11.

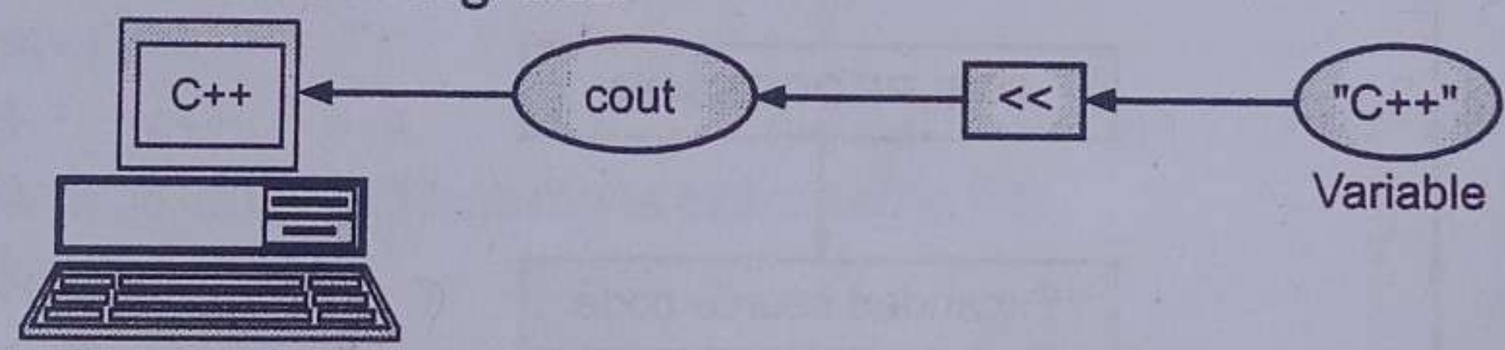


Fig. 1.11: Output with cout Operator

Examples:

```
1. int a;
   cout<<a;
2. float salary;
   cout<<salary;
3. char name[15];
   cout<<name;
```

- We can display more than one outputs called cascaded outputs.

Syntax:

```
cout<<var1<<var2<<.....<<varn;
```

Examples:

```
1. cout<<"x="<<x;
2. cout<<roll<<"  "<<name;
3. cout<<roll<<"\t"<<name;
4. cout<<name<<endl;
```

1.6 COMPILING, LINKING AND RUNNING A C++ PROGRAM

- A C++ compiler is itself a computer program which only job is to convert the C++ program from our form to a form the computer can read and execute.
- The original C++ program is called the **"source code"** and the resulting compiled code produced by the compiler is usually called an **"object file"**.
- Before compilation the preprocessor performs preliminary operations on C++ sources files. Preprocessed form of the source code is sent to compiler.
- After compilation stage object files are combined with predefined libraries by a **linker**, sometimes called a binder, to produce the final complete file that can be executed by the computer.
- A library is a collection of pre-compiled "object code" that provides operations that are done repeatedly by many computer programs.
- The compilation and linking process of a C++ program is shown in Fig. 1.12.
- Linking is the process of taking all the object files for a program and combining them into a single executable.

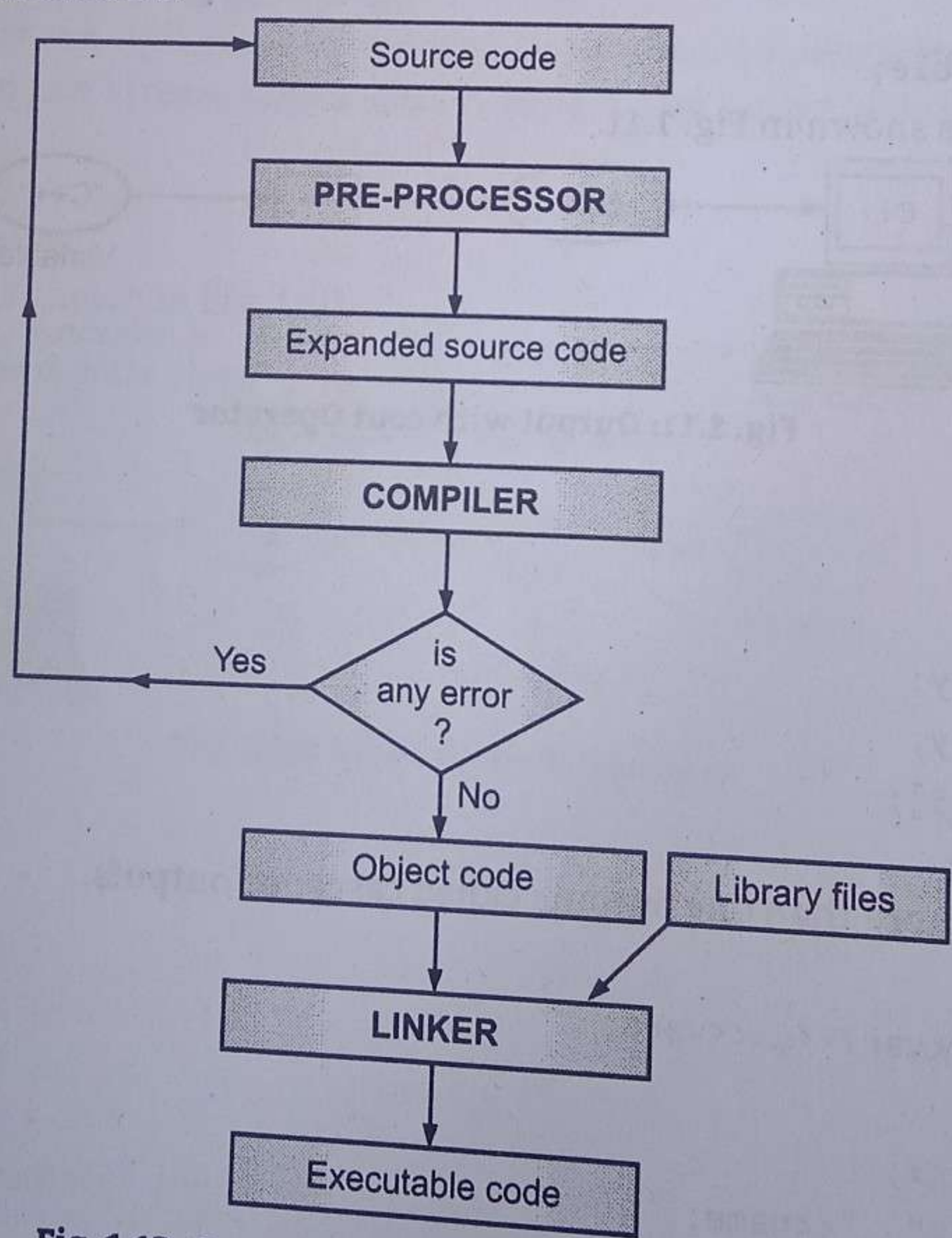


Fig. 1.12: Compilation and Linking Process of C++

- C++ program can compile and execute on MS-DOS or UNIX environment.

1. **On MS-DOS,** operating system we can use turbo C++ compiler and compilation process is same as C language. The C++ program which is having extension. .cpp, is compiled by compiler and source program is translated into object code (.obj, files). These .obj files are processed by a linker to link the necessary system files or library files required for the program. Then executable files (.exe) are created.

   Steps:

   (i) Compile (click on Compile from menu bar) linking is message display on screen while compiling.

   (ii) Run (click on Run from menu bar).

2. **On UNIX, operating system:** We can write source code is Vi editor or any other text editor provided on UNIX. For most UNIX system, C++ compilers are available directly from the system's vendor. If your system does not have a C++ compiler, you may be able to find a copy of GNU C++ that works on your system.

   Steps:

   (i) `g++ - g program name.cpp`

   This will compile C++ source code and a.out is created which is executable file.

   (ii) For all C++ program, the executable file a.out is created. If we want to specify the name for outfile then we use following command on UNIX, prompt.

   ```
   $ g++ - g - o outputfilename programname
   ```

   Where, outputfilename is executable file

   For example:

   ```
   $ g++ - g - o out out.cpp
   ```

   Here, out is executable file.

   (iii) To run the program as follows:

   ```
   $ ./a.out
   ```

   (iv) For debugging a program, we use executable file with gdebugger

   ```
   $ gdb outputfilename
   ```

   For example, `$ gdb a.out`

   or `$ gdb out`

   For stepwise debugging use following command,

   ```
   $ gdb run
   $ gdb break 2
   ```

   ⟶ line number where we want to debug

   ```
   $ s (for step)
   $ quit
   ```

   For more information see manual of gdb using command,

   ```
   $ man gdb
   ```

3. **Turbo C++ Compiler:** The first and frequently used method for creating program is Turbo C++'s Integrated Development Environment (IDE). To start IDE type TC at DOS prompt. Or search the file TC.EXE in our computer and Run it. our IDE will look like this ...

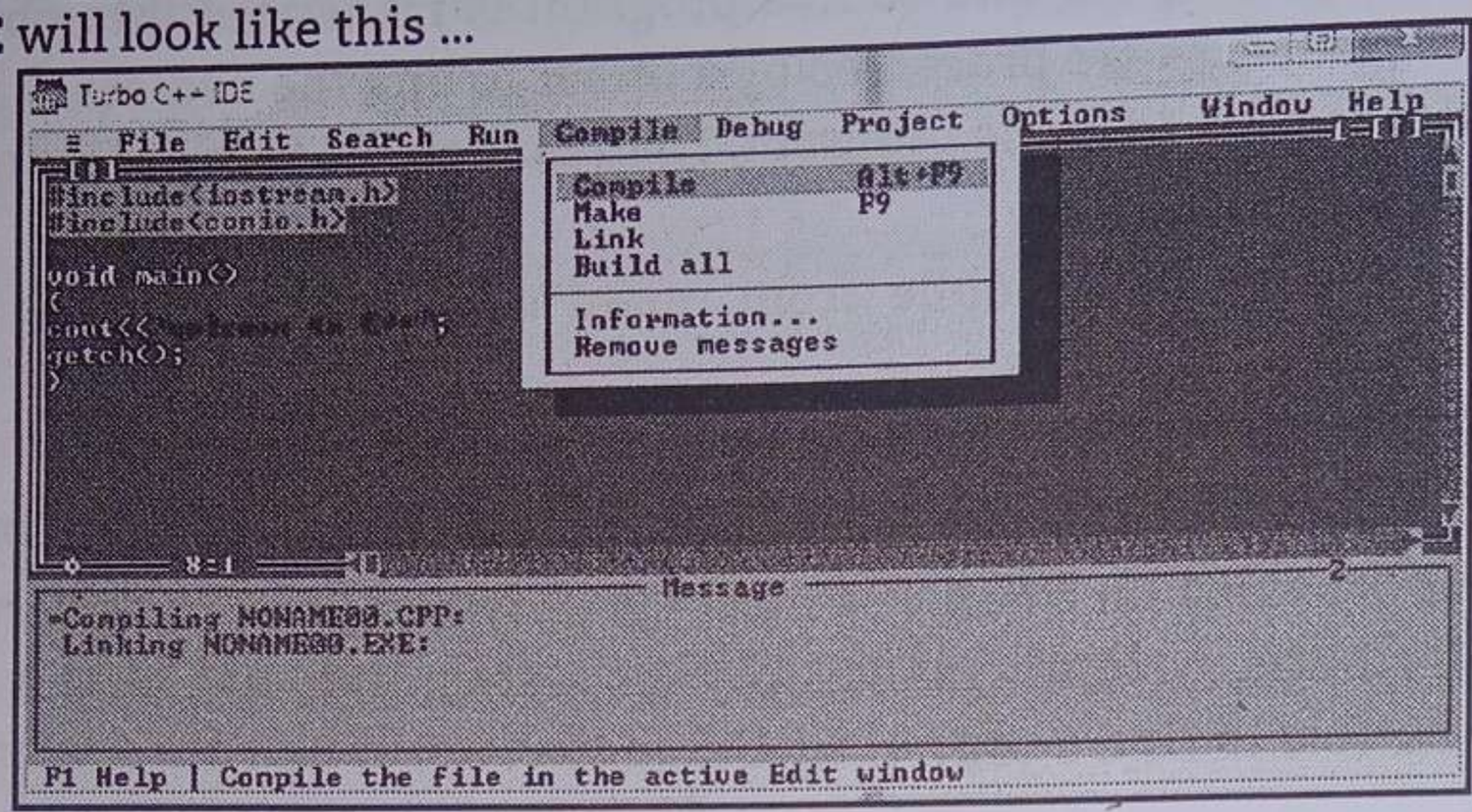


Fig. 1.13: Turbo C++ Compiler

(i) Now type sample program on Editor:
(ii) Click on Compile menu choose Compile option or press Alt+F9.
(iii) Click on Run menu choose Run option or press Ctrl+F9.
(iv) If there is no error output will be displayed on User Screen.

Summary

- C++ was developed by Bjarne Stroustrap in early years 1980's at AT&T Bell Laboratories.
- C++ supports numeric (integer and floating) as well as character and string constants.
- C++ is a superset of C language with added OOP features. C++ adds a number of object oriented features such as objects, inheritance, function overloading and operator overloading to C.
- C++ can be used to build a variety of systems such as editors, compilers, databases, communication systems, and many more complex real-life application systems.
- Like C programs, execution of all C++ programs begins at main() function and ends at return() statement. The header file iostream should be included at the beginning of all programs that use input/output operations.
- All ANSI C++ programs must include using namespace std directive.
- A typical C++ program would contain four basic sections, namely, include files section, class declaration section, member function section and main function section.
- Like C programs, C++ programs can be created using any text editor like DosBox.
- Most compiler systems provide an integrated environment for developing and executing programs.
- Popular systems are UNIX AT & T C++, Turbo C++ and Microsoft Visual C++.
- C++ supports single-line (//) and multi-line (/*....*/) comments. All characters available inside any
- Comment are ignored by C++ compiler.

Check Your Understanding

1. Comments are included in programs to provide explanations for human readers; and it is denoted by _____.
   (a) / (b) //
   (c) * (d) **
2. C++ requires every statement to end with a _____.
   (a) Colon (b) Question mark?
   (c) Comma (d) Semi-colon
3. C++ is an extension of C language but C++ is a general purpose _____.
   (a) Markup language
   (b) Scripting language
   (c) Object Oriented programming Language
   (d) Tags based programming language
4. In C++, the semicolon is used as a _____.
   (a) Statement terminator (b) Newline symbol
   (c) Statement operator (d) None of them
5. C++ was developed by _____.
   (a) Dennis Ritchie (b) Ada Lovelace
   (c) Bjarne Stroustrup (d) James Gosling
6. What does #include means in C++ programs?
   (a) It's a directive
   (b) It is used for copy and import
   (c) It causes the contents of a second file to be inserted into the original file
   (d) All of them
7. C++ is a _____.
   (a) Case-sensitive language (b) Case-insensitive language
   (c) Only lowercase letter identifier (d) None of them
8. The standard input and output library of C++ is _____.
   (a) stdio.h (b) istream.h
   (c) iostream.h (d) conio.h
9. The main() function in the header of a program indicates what _____.
   (a) It tells the compiler that where the program begins
   (b) It tells the compiler that it's a C++ program
   (c) It tells the compiler that a comment is in the code
   (d) It indicates nothing

Answers

1. (b)	2. (d)	3. (c)	4. (c)	5. (c)	6. (d)	7. (a)	8. (c)	9. (a)

Practice Questions

Q.I Answer the following questions in short:

1. What is C++?
2. List out procedure oriented programing language.
3. What is Compiling?
4. Define Linking.
5. Who is developer of C++ language?
6. What is input stream?
7. What is output stream?

Q.II Answer the following questions:

1. With the help of diagram describe evolution of C++.
2. Compare POP and OOP.
3. Explain Compiling and Linking process of C++ program.
4. List various features of C++.
5. Write short note on: Anatomy of C++ Program.
6. How C++ evolved from C?

Q.III Define terms:

1. Comment
2. Compile
3. Linking
4. Input stream
5. Output stream

Previous Exam Questions

Summer 2018

1. What is difference between OOP & POP? **[1 M]**

Ans. Refer to Section 1.4.3.

Winter 2018

1. In object oriented programming focus is on: **[1 M]**

(a) Data
(b) Procedure
(c) Functions
(d) Inheritance

Ans. (a) Data

2... Object Oriented Programming Concepts

Objectives...

- To understand concepts of Classes and Objects.
- To study about Abstraction and Inheritance.
- To learn Polymorphism and Data Binding.
- To study Encapsulation and Message Passing.

2.1 INTRODUCTION

- During the late 1970's and early 1980's, 'C' had become very popular language. C was a structural programming language. However, for large and complex program, the structural approach failed to show the desired results in terms of bug-free, easy-to-maintain and reusable programs. To deal with this a new way to program was invented "Object Oriented Programming".
- Object Oriented Programming (OOP) is playing an increasingly significant role in the analysis, design and implementation of software systems. OOP languages provide the programmer the ability to create class hierarchies, instantiate objects and send messages between objects to process themselves.
- Object oriented means to organize software as a collection of discrete, real-world objects that incorporate both data structure and behavior.
- Object Oriented Modeling and Design (OOMD) is a new way of thinking about problems using models organized around real-world concepts. The fundamental construct is the object, which combines both data structure and behavior in a single entity.
- Object oriented models are useful for understanding problems, communicating with application experts, modeling enterprises, preparing documentation and designing programs and databases.

- Object oriented development is a conceptual process independent of a programming language until the final stage.
- Object Oriented Programming is a new way of organizing code and data that promises increased control over the complexity of software development process.
- The prime purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming languages.

2.1.1 Need of OOP

- The use of structured programming made more complex programs, less complex and easier to understand. But even with structured programming, when a program reaches a certain size, its complexity exceeds that which a programmer can manage.
- To deal with this approach, a new way to program, was invented **"Object Oriented Programming"**. This method of programming reduces the program complexity by incorporate rating features like inheritance, encapsulation and polymorphism.
- OOP is a way of programming help the programmer to comprehend and manage large and complex programs.
- Object Oriented Programming models real-world objects with software counterparts. It takes advantage of class relationships where objects of certain class have the same characteristics.
- Object Oriented Programming (OOP) allows us to decompose a problem into a number of entities called objects.
- OOP takes advantages of inheritance and multiple inheritance relationship where newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own.
- OOP gives us a more natural way to view the programming process, namely by modeling real-world objects. C++ was the first object oriented programming language. It was invented by **"Bjarne Stroustrup"**. This language was initially called as **"C with classes"**.

2.2 OBJECT ORIENTED CONCEPTS

- In this section we study classes and objects in detail.

2.2.1 Classes

Class Definition:

- Class is a user defined data type with data elements and functions (operations).
- A class is a way that binds data and functions that operate on the data together in a single unit. Like other user defined data types, it also needs to be defined before using its objects in the program.

- A class definition specifies a new data type that can be treated as a built-in data type.
- The data members describe the state of the object. The operations defines the behaviour of the object.
- Class is used to define abstractions. A class contains access specifiers, data members and member functions.

Declaration of Class:

- The keyword 'class' is used to declare a class in C++. The declaration of a class always ends with the semicolon (;). All data members and member functions are declared within the braces { and } which are called the body a class.
- **Syntax for declaration of a class:**

```
class class_name
{
   // body of a class
};
```

- The class specifies type and scope of its members. The keyword *class* indicates that the class name which follows in an abstract data type. The body of a class is enclosed within the curly braces followed by a semicolon (;). The body of a class contains declaration of variables and functions, collectively known as data members and functions are known as member functions. This members are grouped under three section, private, protected and public which defines the visibility of members.
- Consider the following simple class example:

```
class student
{
   int rollno;
   char name[20];
   public:
      void getdata()
      {
         rollno = 10;
         name = "OMKAR";
      }
      void display()
      {
         cout<<"Roll no:"<<rollno;
         cout<<"Name:"<<name;
      }
};  //end of class
```

- A class called student contains two data members and two member functions.
- The data members are private by default whereas both the member functions are public.
- The general notation of representation of class student is shown in Fig. 2.1.

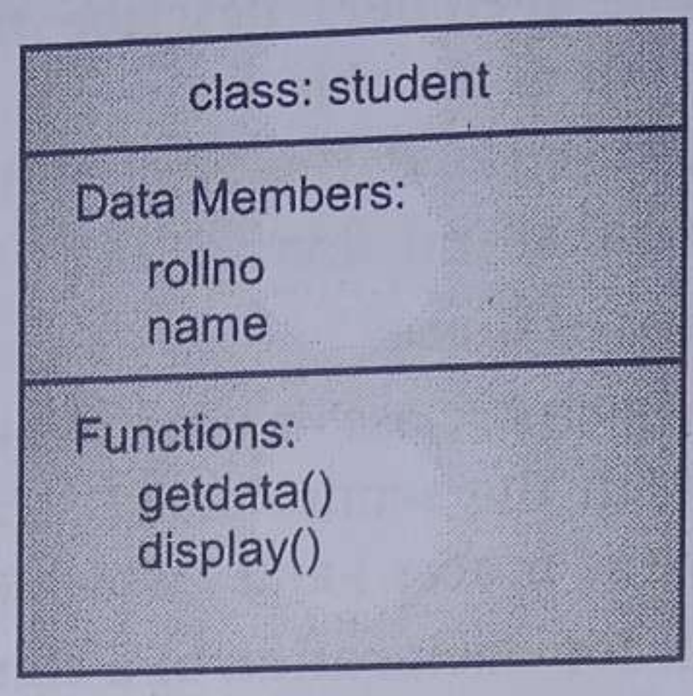


Fig. 2.1: Class

2.2.2 Objects

- An object is an instance of a class. A class provides blueprints for the object. An object is a variable of type class.
- Objects are discrete, distinguishable entities. Objects are basic run-time entities in an OOP environment. Any real word object can be treated as an object in the OOP such as a paragraph in a document or a window on a workstation or the white queen in a chess game are some examples of objects.
- Objects can be concrete such as a file in a file system or conceptual such as scheduling policy in a multiprocessing operating system. Each object has its own inherent identity.
- Objects serve two purposes i.e., they promote understanding of the real-world; provide a practical basis for computer implementation and decomposition of a problem into objects depends on judgments and the nature of the problem.
- Program objects should be chosen such that they match closely with the real-world objects.
- In a program, objects interact by sending messages to one another. Each object contains data and code to manipulate the data.
- Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted and the type of response returned by the objects.
- Objects in a class have the same attributes. Objects derive their individuality from differences in their attribute values and relationships to other objects. For example, table, chair, desk are members of class furniture.
- In our day today life, we come across a number of objects. For example, PC, television, radio, pen, pencil, stack, queue, program file, table, car, telephone, flower, mango etc. as shown in Fig. 2.2.
- An object will have the following two characteristics:
  (i) State or attribute.
  (ii) Behavior or operation.

- The state or attribute tells the built-in characteristics of an object. They remain the same till disturbed or modified. For example, a flower has attributes such as, Shape, Color, Smell and so on.
- The behavior or operation of an object tells about its action. It can be explicitly defined. For example, flower blossoms, bud etc.
- Therefore, objects are the basic run-time entities like represent a person, a place, a bank account, a table of data or any item that the program has to handle.
- Some objects may correspond to real-world entities such as students, employees, inventory items whereas others may correspond to computer hardware and software components.

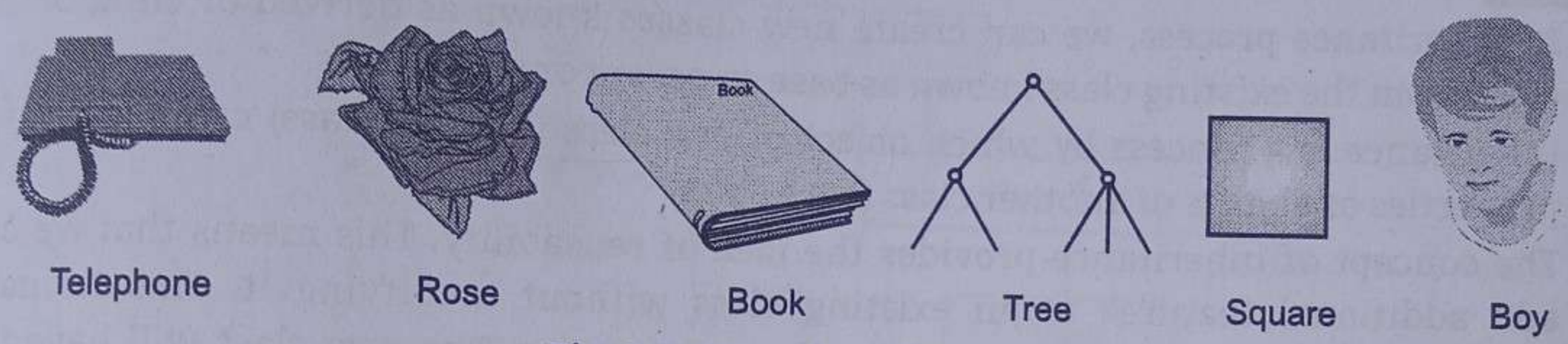


Fig. 2.2: Examples of Objects

- When we create a object of a class:
  - Memory is allocated.
  - Constructor is called implicitly.
  - Memory gets initialized.
- The process of creating object is called class instantiation. We can create object as:

```
class_name objectname;
```

OR

```
class_name  object1, object2, ........ objectn;
```

- **For example:**

```
student   S;
student   S1, S2;    //more than one object of class student
```

- The another way is, we can create object by placing their name immediately after the class declaration.

```
class student
{
    ..........
    ..........
    ..........
} S1, S2;
```

2.2.3 Data Abstraction

- Abstraction is the selective examination of certain aspects of a problem.
- The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant.

- In other words, we can say that abstraction is the act of representing essential features without including the background details or explanations.
- Data abstraction is the process of defining a data type, often called Abstract Data Type (ADT), together with the principle of data hiding.
- The definition of an ADT involves specifying the internal representation of the ADTs data as well as the functions to be used by others to manipulate the ADT.
- In OOP, classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost and functions to operate these attributes. Classes are also known as ADTs.

2.2.4 Inheritance

- In inheritance process, we can create new classes known as derived or child or sub class from the existing class known as base or parent or super class.
- Inheritance is a process by which object of one class (derived class) can acquire the properties of objects of another class (base class).
- 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 made possible by deriving a new class from the existing one. The new class will have the combined features of both the classes.
- For example, plant is base, through which two types like flowering and non-flowering is derived.

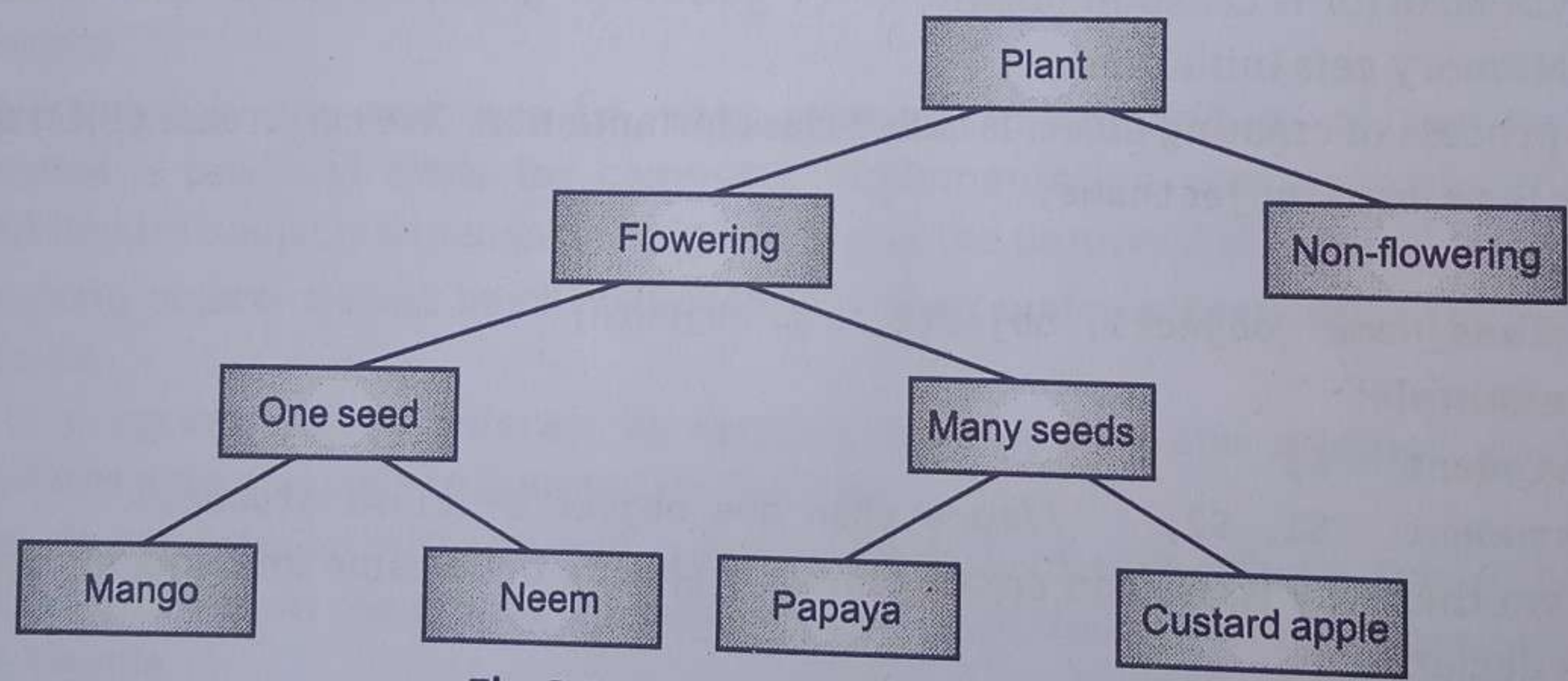


Fig. 2.3: Example of Inheritance

- The new class will have the combined features of base class. The basic advantage of using inheritance is to increase reliability and decrease in maintenance cost. Because the base and derived classes can be shared by many users.

2.2.5 Polymorphism

- In polymorphism "poly" means many and "orphism" means form, which means many forms of one things.
- In other words, "Polymorphism means one thing different or many forms" i.e. the ability to take more than one form.

- Polymorphism plays an important role in allowing objects having different internal structures to share the same external interfaces.
- In OOP, polymorphism refers to the fact that a single operation can have different behaviour in different objects. In other words, different objects react differently to the same message.
- For example, consider the operation of addition. For two numbers, the addition should generate the sum. The operation of addition is expressed by a single operator +. You can use the expression x + y to denote the sum of x and y for many different types of x and y integers, floating point numbers and complex numbers and even the concatenation of two strings.
- The types of polymorphism are shown in Fig. 2.4.

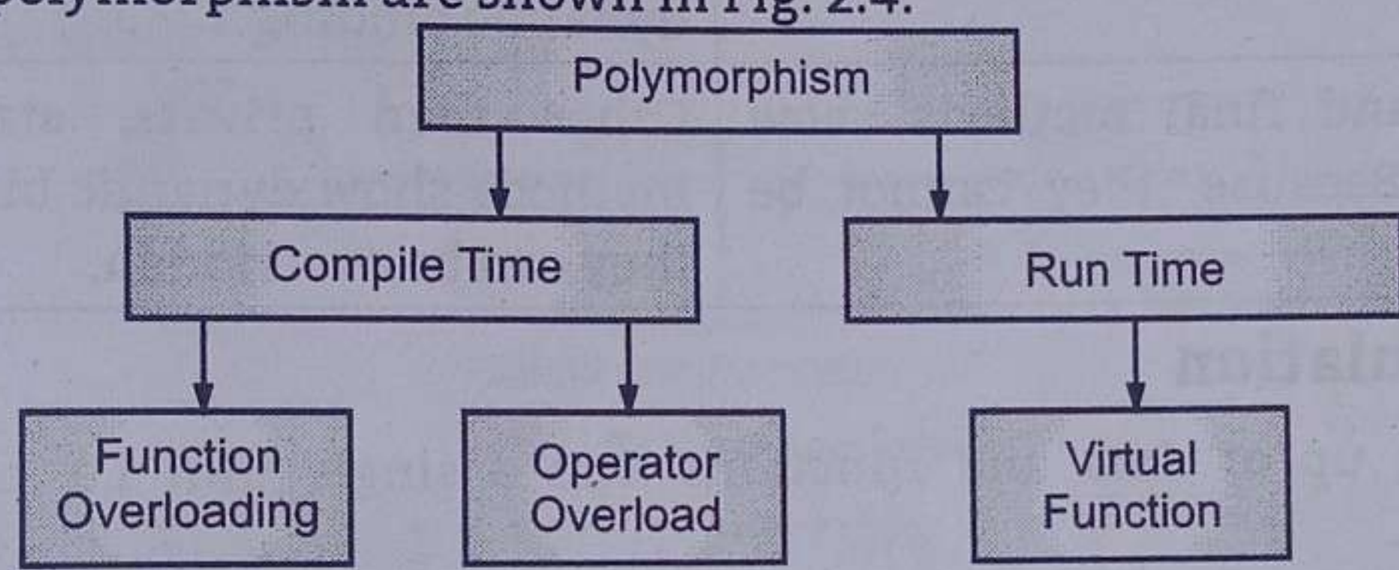


Fig. 2.4: Types of Polymorphism

- Compile time polymorphism is also called as **Early Binding** or **Static Binding**.
- Run time polymorphism is also called as **Late Binding** or **Dynamic Binding**.

2.2.6 Data Binding

- There are two types of Binding.
  1. Static Binding (Early Binding)
  2. Dynamic Binding (Late Binding)

1. Static Binding:

- Binding generally means linking. In a C++ Program choosing a function in a normal way during its compilation time is called as Static Binding.
- The static binding is also referred as early binding or static linkage. We can say that the static binding an example of compile time binding.
- Method Overloading happens at compile time (Static Binding). In method overloading, method call to its definition at compile time (Static Binding).

2. Dynamic Binding:

- Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. Dynamic binding also known as late binding.
- Dynamic binding is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.
- Overriding happens at runtime (Late Binding). In method overriding, method call to its definition at runtime (Dynamic Binding).

Difference between static binding and dynamic binding:

Table 2.1: Difference between Static Binding and Dynamic Binding

Static Binding	Dynamic Binding
It is a binding that happens at compile time.	It is a binding that happens at run time.
Actual object is not used for binding.	Actual object is used for binding.
It is also called early binding because binding happens during compilation.	It is also called late binding because binding happens at run time.
Method overloading is the best example of static binding.	Method overriding is the best example of dynamic binding.
Private, static and final methods show static binding. Because, they cannot be overridden.	Other than private, static and final methods show dynamic binding. Because, they can be overridden.

[W-18]

2.2.7 Encapsulation

- The wrapping up of data and functions into a single unit i.e. class is called as Encapsulation.
- In other words, "the binding of data and function into a single unit (class) is called as encapsulation".
- The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.
- These functions which are wrapped in the class provide the interface between the objects data and the program.
- Encapsulation is a mechanism that keeps the data and code safe from external interference and misuse. This insulation of the data from direct access by the program is called data hiding which is also known as data encapsulation.

2.2.8 Message Passing

- An OOP consists of a set of objects. These objects communicate with each other. The process of programming in an object oriented language, therefore, involves the following three basic steps:

  Step 1 : Create class which define object and their behaviour.

  Step 2 : Create object from class definitions.

  Step 3 : Establish communication between objects.
- Objects communicate with one another by sending and receiving information. This is same as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

- The message for an object is a request for execution of a procedure and therefore will invoke a function, (procedure) in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function, (message) and the information to be sent.

Example:

```
employee.salary (name);
```

Object Message Information

- Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as along as it is alive. The another way of representing message passing is shown in Fig. 2.5.

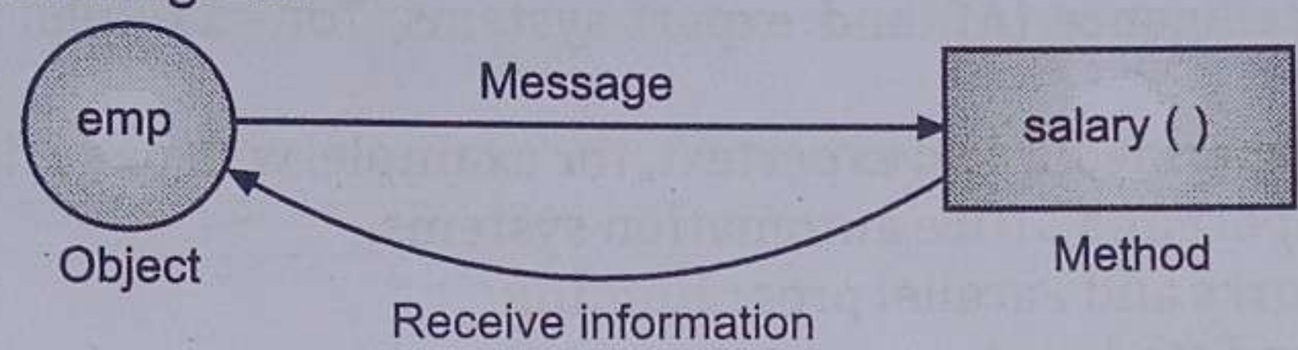


Fig. 2.5: Message Passing

2.3 ADVANTAGES AND APPLICATIONS OF OOP

2.3.1 Advantages of OOP

- The object oriented approach focuses on identifying objects from the application domain and then fitting procedures around them. It is based on underlying framework of the application domain. Its object orientation contributes to the solution of many problems associated with the development and quality of software products.
- The advantages of OOP are listed below:
  1. Data abstraction concept consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental properties. The use of abstraction preserves the freedom to make decisions as long as possible.
  2. By the use of property inheritance, we can eliminate redundant code and extend the use of existing class.
  3. The property of data encapsulation helps to programmer to build secure programs that cannot be invaded by code in other parts of the program.
  4. We can have multiple instances of an object to co-exist without any interference.
  5. We can map objects in the problem domain to the object in the program.
  6. We can modularise the program by dividing it into classes and objects.
  7. The data centered design approach enables to capture more details of a model in implementable form.
  8. Interface descriptions with external systems become much simpler by the property of message passing i.e. the communication between objects.
  9. Object oriented programming supports reusability. Reusable software reduces design, coding and testing cost by amortizing effort over several design. Reducing the amount of code also simplifies understanding.

10. In OOP, complex software's can be easily managed.
11. OOP saves development time and gives higher productivity.
12. In OOP, small projects (programs) can be easily upgraded to larger one.
13. Message passing in OOP provides the interface with external systems. [W-18, S-19]

2.3.2 Applications of OOP

- Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of applications because it can simplify a complex problem.
- The areas for application of OOP includes:
  1. Real-time systems, for example, oil exploration plant.
  2. CAM/CAD systems.
  3. Object-oriented databases.
  4. Artificial Intelligence (AI) and expert systems, for example, Disease knowledge base.
  5. Hypertext, hypermedia and expertext, for example, web page designing.
  6. Decision support and office automation systems.
  7. Neural networks and Parallel programming.
  8. Simulation and Modeling.

Summary

- OOP stands for Object Oriented Programming. OOP is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.
- Example of OOP are C++, Java, Eiffel, Smalltalk, VB.Net, C#.Net etc.
- Basic object oriented concepts are objects, classes, encapsulation, inheritance, polymorphism.
- Object is the basic unit of object oriented programming. In OOP, a problem is considered as a collection of a number of entities called objects. Objects are instances of classes.
- In OOP a class gives the blueprint for a set of similar objects. A class is a user defined type consisting of data members and member functions.
- Data abstraction increases the power of programming language by creating user defined data types. Data abstraction also represents the needed information in the program without presenting the details.
- In OOP, insulation of data from direct access by the program is called data hiding.
- Inheritance is the process of forming a new class from an existing class or base class. The base class is also known as parent class or super class. The new class that is formed is called derived class. Derived class is also known as a child class or sub class.
- Data encapsulation combines data and functions into a single unit called class. When using data encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class.
- Data encapsulation enables the important concept of data hiding possible. Inheritance helps in reducing the overall code size of the program.

- In OOP, polymorphism means one name, multiple forms. It allows us to have more than one function with the same name in a program.
- Binding refers to the linking of a procedure call to the code to be executed in response to the call. When memory is allocated at run-time it is known as dynamic binding.
- In OOP, message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.
- Object oriented approach has many benefits like Ease in software design, Ease in software maintenance and Reusability of software.
- Applications of OOP has gained importance in almost all areas of computing including real-time business systems.

Check Your Understanding

1. The word "oriented" means showing _____.
   (a) ambiguity (b) functionality
   (c) methodology (d) none
2. OOP decompose the problem into smaller unit known as _____.
   (a) object (b) class
   (c) function (d) none
3. Object-oriented programming language gives emphasis on _____.
   (a) flowchart (b) algorithm
   (c) data (d) none
4. Object-oriented programming language follows the _____ approach.
   (a) top-down (b) bottom-up
   (c) top-bottom (d) none
5. The function that operate on the data of an object are tied together in the data structure, this concept is known as _____.
   (a) Polymorphism (b) inheritance
   (c) encapsulation (d) none
6. _____ allows creation of new classes from old one.
   (a) Polymorphism (b) inheritance
   (c) class (d) none
7. _____ are basic run-time entities.
   (a) data (b) classes
   (c) objects (d) none
8. The object with the attributes and methods are grouped together to form a _____.
   (a) data (b) class
   (c) object (d) none
9. An object is an _____ of a class.
   (a) abstraction (b) operation
   (c) instance (d) none
10. The code redundancy can be eliminated through _____.
   (a) polymorphism (b) inheritance
   (c) class (d) none

Answers

1. (b)	2. (a)	3. (c)	4. (b)	5. (c)	6. (b)	7. (c)	8. (b)	9. (c)	10. (b)

Practice Questions

Q.I Answer the following questions in short:

1. State any one advantage of object oriented programming over procedure oriented programming.
2. C++ is a pure object oriented language. Justify true or false.
3. What do you mean by data encapsulation and data hiding?
4. What is static binding and dynamic binding?
5. List out object oriented concepts.

Q.II Answer the following questions:

1. Write short note on:
   (a) Objects, (b) classes, (c) Data abstraction, (d) data encapsulation, (e) Inheritance, (f) polymorphism
2. Describe inheritance as applied to OOP.
3. State the advantages of OOP.
4. State the applications of OOP.
6. What is difference between static binding and dynamic binding.
7. Write short note on need for OOP.
8. Explain any 4 object oriented concepts in detail.

Q.III Define terms.

1. Encapsulation
2. Data abstraction
3. Polymorphism
4. Object
5. Classes
6. Inheritance

Previous Exam Questions

Summer 2018

1. By default members of class are _____. **[1 M]**
(i) protected, (ii) private, (iii) public, (iv) all of above

Ans. (ii) private

Winter 2018

1. The insulation of data from direct access by unauthorized functions is called _____. **[1 M]**
(a) inheritance, (b) data hiding, (c) polymorphism, (d) message passing

Ans. (b) data hiding

2. State the advantage of encapsulation. **[1 M]**

Ans. Refer to Section 2.2.7.

3. List any four applications of OOP. **[1 M]**

Ans. Refer to Section 2.3.2.

Summer 2019

1. List the applications of object oriented programming. **[1 M]**

Ans. Refer to Section 2.3.2.

3... Programming in C++

Objectives...

- To study different Data Types, Operators and Keywords.
- To learn about Type casting in C++.
- To study Reference Variables, Classes and Access Specifiers.
- To learn how to Defining Data Members and Member Functions.
- To understand Usage of namespace and Usage of Manipulators.

3.1 INTRODUCTION

- The C++ program is made up of sequence of characters. The characters are the most basic building blocks of the C++ language. Using only these characters, the language constructs are formed.
- Character set is a set of valid characters recognised by a language. A character represents any letter, digit, or any other sign.
- The C++ character set consists of upper and lower case letters, digits, special characters and white spaces. The letters and digits together constitute the alphanumeric set.
- Token is a sequence of characters from the character set of C++, which is identified by the compiler.
- Keywords are used for specific purposes in C++ and compiler can interprets these words.

3.2 DATA TYPES

- A data type determines the type and the operations that can be performed on the data. A data type is defined as, "the set/collection of values and the set of operations that operate on those values".
- In short, data type is analyzed as follows:

  Data Type = Permitted Data Values + Operations
- The various data types provided by C++ are built-in data types, derived data types and user-defined data types as shown in Fig. 3.1.

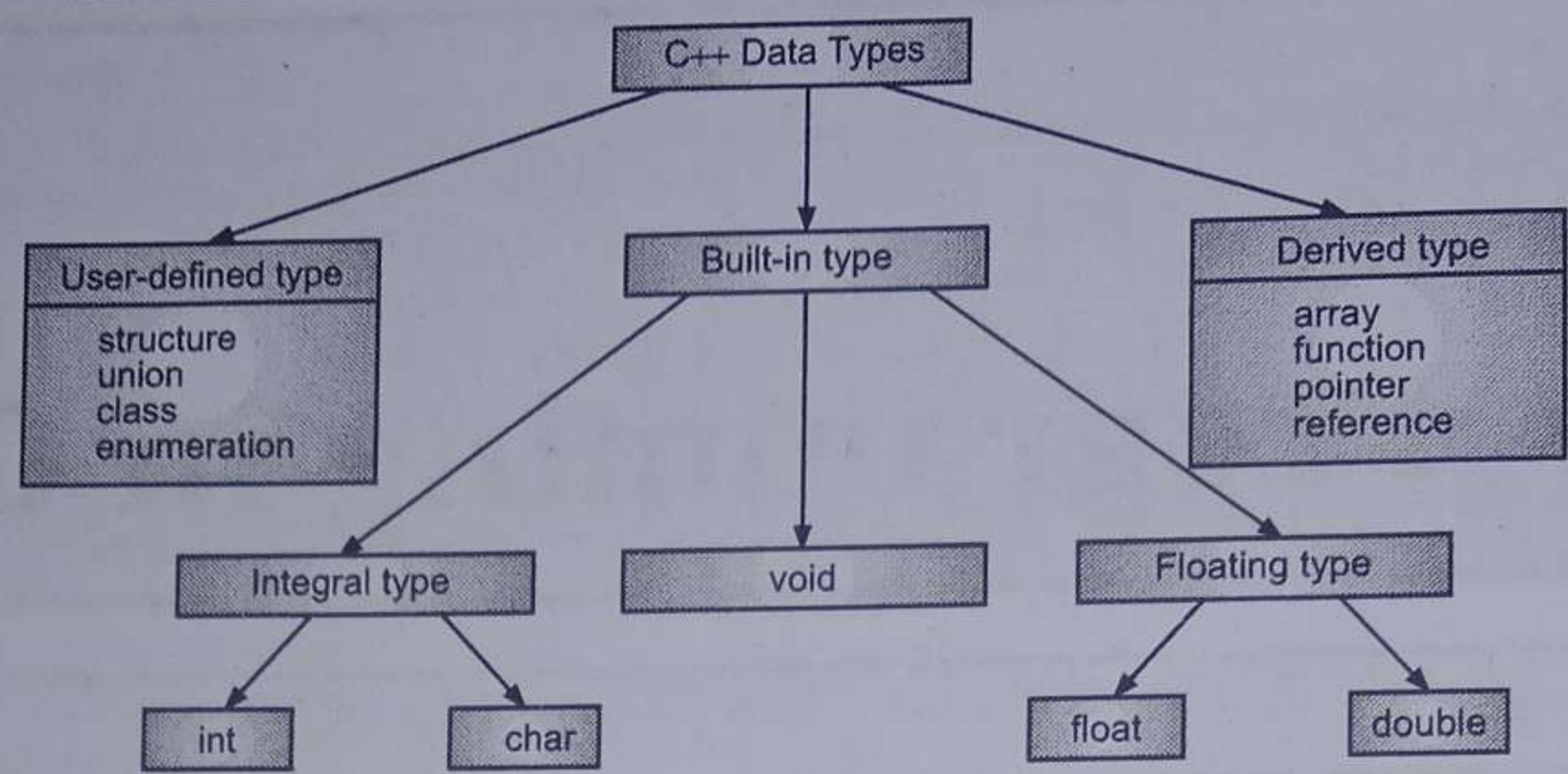


Fig. 3.1: Hierarchy of C++ Data Types

1. Built-in Data Types:

- Built-in data types are atomic data type which is not composed of other data types. C++ compilers support all the built-in (also known as basic or fundamental) data types as shown in Fig. 3.1.
- C++ basic data types are integer, float, character and boolean. C++ uses two more data types, bool and wchar_t.
- The Table 3.1 shows the description of C++ basic data types.

Table 3.1: Basic Data Type, Size and Ranges

Type	Bytes	Range
char	1	signed char: – 128 to 127 unsigned char: 0 to 255
int	2	signed int: – 32768 to 32767 unsigned int: 0 to 655535
short int	2	signed: – 32768 to 32767 unsigned: 0 to 655535
long int	4	signed: – 2147483648 to 2147483647 unsigned: 0 to 4294967295
float	4	3.4 E – 38 to 3.4 E + 38
double	8	1.7 E – 308 to 1.7 E + 308
long double	10	3.4 E – 4932 to 1.1 E + 4932 (Compiler dependent)

- The short, long and unsigned qualifiers are used with int types. We can used unsigned qualifier when we want the variable to have positive values only.

For example,

```
short number;     //short int
long area;        //long int
unsigned i,j;     //unsigned int
double result;
```

2. User-defined Data Types:

- User has the facility in the programming language to create data types according to his/her own needs.
- The data types defined by user according to his/her requirements known as user-defined data types.

(i) Enumerated:

- In C++, enumerated data type helps in enhancing the reliability of the code. Enumerations are a way of defining constants.
- **Syntax:**

```
enum tag {member1, member2, ........, membern};
```

- Here, enum is the keyword and tag is the name of enumeration and member1, member2, ... membern are identifiers which represents integer constant values. The member should be unique. Once, the enumeration is define then variable or object of that type is defined.

Syntax:

```
enum tag {member1, member2 ......} variables;   OR
enum tag {member1, member2 ......} object_name;
```

- Here, all members take integer values starting with zero i.e. member1 is assigned 0, member2 is assigned 1 ..... and so on.

For example,

```
enum colors {black, green, red, yellow, blue};
enum colors background;
Here,
black = 0
green = 1
red = 2
.
.
.
```

- Enumeration variables are particularly useful as flags to indicate various options are to identify various conditions. It increases clarity of the program. If we want to have blue colors for the background, it is easier to understand with,

```
background = blue;
```

Than with,

```
background = 4;
```

It makes program more readable.

(ii) Structure:

- In 'C' structure is a collection of data items of different data types. Structure in C++ is a collection of variables of different data types in a single name.

- **Syntax:**

```
struct name of structure
{
    data members;
    functions
}
```

- For example,

```
struct student
{
    private;
        char name[20];
        long phone;
    public:
        void display(void)
            {
                cout<<phone;
            }
};
```

- By default members of the C++ structures are public.

For example,

```
struct student s;
printf ("%ld",  s.phone);     //Invalid
```

- We cannot print phone like this, we have to invoke public function display to print phone i.e. Instead of printf statement we can write,

```
s.display();
```

(iii) Union:

- Union in C++ is a user defined data type that uses the same memory as other objects from a list of objects. At an instance it contains only a single object.
- **Syntax:**

```
union tag
{
    member1;
    member2;
    '
    '
    member n;
} objectname;
```

- **For example:**

```
union example
{
    int a;
    char b;
    float c;
} e;
```

- There are three elements of different types int, char and float respectively. Here, int takes two bytes memory, char takes one byte and float takes four bytes. Union allows the same storage is shared with its members. Hence, total memory required is four bytes. Unions are very useful for applications involving multiple members, where values need not be assigned to all members at one time. While in case of structure for same example, memory required is 2 + 1 + 4 = 7 bytes to store its contents.

(iv) Class:

- C++ also permits to define another user-defined data type known as class which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the focus of object oriented programming. Generally, class is used in C++ instead of struct. Class is user define data type with data elements and functions.
- **Difference between a class and a structure in C++:** There is not much difference between a class in C++ and a structure in C++. Everything which is defined in structure in C++ is also applicable to the classes in C++.
- **The two differences are:**
  - The keyword class is used instead of struct.
  - All the members in a class are by default private.
- For example,

```
class student
{
    int roll;
    char name[20];      } By default both are private
    public:
        void getdata(void);
        void putdata();
};
```

3. Derived Data Types:

- The data types, which are extracted/derived from fundamental data types are called derived data types.

(i) **Arrays:** An array in C++ is a collection of fixed-size finite number of objects, data value that belong to the same type. An array is a collection of data storage locations. Each of the locations of an array in C++ holds same type of data. The storage location is called an element or component of the array. The individual element within the array is called subscript. In C, array size is exact same length of the string constant.

For example: `char string[5] = 'abcde";//valid in C`

But in C++, the array size is one larger than the number of characters present in the string. Here, "\0" is treated as a one character.

For example:
```
char string[5] = "abcde";   //Invalid in C++
char string[5] = "abcd";    //Valid in C++
```

(ii) Functions: A function is a discrete block of statements that perform certain task. Once, a function has been written to play a particular role it can be called upon repeatedly throughout the program. Function is the named part of a program that can be invoked from other parts of the program as and when required. A function must be declared before its use, such a declaration is known as a function prototype it ends with semicolon.

Syntax: `return_type function_name (parameters);`

For example: `int max (int x, int y);`

(iii) Pointers: A variable in C++ that holds a memory address is called a pointer. This address is usually the location of another variable in memory. If one variable contains the address of another variable, the first variable is said to point to the second. A pointer declaration consists of a base type, an * (asterisk) and the variable name.

Syntax: `datatype * ptvar;`

Where, ptvar is pointer variable.

For example: `float * i;  //float pointer`

Program 3.1: Write a program which demonstrates the declaration of pointer.

```
#include<iostream>
using namespace std;
main()
{  int l = 2;
   int *m;
   m = &l;
   printf ("\n Address of l = %u", m);
   printf("\nValue of l = %d", *m);
}
```

Output:

```
Address of l = 6242
Value of l = 2
```

Accessing variable through its pointer:

- Once, a pointer has been assigned the address of a variable then we can access the value of the variable using the pointer. If the above Program 3.1, we add a statement,

  `c = *m`

  Where, c is int and it access value of address.

  `*m = 30; //30 assigned to l through indirection.`

- In addition to this C++ also have constant pointer declaration and pointer to constant declaration.

  For example:

  ```
  int *const m = "xyz";  //constant pointer
  int const *m = &l;     //pointer to a constant
  ```

  In constant pointer we cannot change address of 'm' and in pointer to a constant, the contents of '*l*' cannot change.

void Pointers:

- void pointers used to point the other data type of variable in C++.

  If we define,

```
int *m;
float *l;
m = &l;    //results in compilation error
```

  For such type compatibility void pointers are used.

  For example,

```
void *v;
int *i;
float *f;
v = &i;
v = &f;
```

  Here, void pointer v will hold pointer of any type.

void Data Type:

- The void data type also known as empty data type. void is used for following two purposes:
  1. If the function is not having any arguments, `functionname(void);`
  2. To specify the return type of function when function is not returning any value.

```
void functionname(void);
```

3.3 OPERATORS IN C++

- In C++, operators can be classified into various categories:

1. Arithmetic Operators:

Table 3.2: Arithmetic Operators

+ (addition)	– (subtraction)
* (multiplication)	/ (division)
% (remainder)	

2. Relational Operators:

Table 3.3: Relational Operators

< (less than)	> (greater than)
<= (less than equal to)	>= (greater than equal to)
== (equal to)	!= (Not equal to)

3. Logical Operators:

Table 3.4: Logical Operators

&&	AND
||	OR
!	NOT

4. Assignment Operators:

Table 3.5: Assignment Operators

=	Assignment
+ =	Addition and assignment
− =	Subtraction and assignment
* =	Multiplication and assignment
/=	Division and assignment
%=	Mod and assignment
&=	Bitwise AND and assignment
^=	Bitwise OR and assignment
<<=	Left shift and assignment
>>=	Right shift and assignment

5. Unary Operators: `-, ++, --, size of, !`

6. Conditional Operator:

- This is the only ternary operator used in C and C++. It is defined as (?:) where the first symbol is ? and second is : (colon).
- **Syntax:**

```
condition expression? expression 1: expression 2;
```

- The condition expression is evaluated first and if true then expression 1 is evaluated otherwise expression 2 is evaluated.
- For example,

```
mfloat a = 15;
float b, c;
a < 20)? b = 50: c = 100;
```

- All these above operators are valid in C and C++.

3.3.1 New Operators in C++

- C++ have some other new operators also. These are:

endl	Line feed operator
setw	Field width operator
new	Memory allocation operator

delete	Memory release operator
::	Scope resolution operator
::*	Pointer-to-member declaration
→*	Pointer-to-member operator
.*	Pointer-to-member operator

- C++ also provides new meaning of some built-in operators. This is operator overloading.

1. Scope Resolution Operator (::) **[S-18, W-18]**

- The scope resolution operator is denoted by a pair of colons (::). It is use to access global variable even if the local variable have the same name as global.
- When the same function name is used for many classes, then this complexity is solved by using scope resolution operator to indicate the class with which the function is associated.

Program 3.2: Simple scope resolution operator.

```
#include<iostream>
using namespace std;
int a=10;
int main()
{
   int a=15;
   cout<<"\n Local a="<<a<<"Global a="<<::a;
   ::a=20;
   cout<<"\n Local a="<<a<<"Global a="<<::a;
}
```

Output:

```
Local a=15 Global a=10
Local a=15 Global a=20
```

- Suppose in a C program, there are two variables with the same name a. One is global (outside a function) and other is local (inside a function). We attempt to access the variable a in the function, we always access the local variable (priority to local). C++ allows the flexibility of accessing both the variables using scope resolution operator (::).

Program 3.3: Use of scope resolution operator for class.

```
#include<iostream>
using namespace std;
class T
{
   int a,b;
   public:
      void getdata();
      void putdata();
};
```

```
void T::getdata()
{
    cin>>a>>b;
}
void T::putdata()
{
    cout<<"a="<<a<<endl;
    cout<<"b="<<b<<endl;
}
void main()
{
    T a;
    a.getdata();
    a.putdata();
}
```

Output:

```
10 20 //input
a = 10
b = 20
```

2. Memory Dereferencing Operators:

- C++ uses pointers to access class members i.e. data and functions. Three memory dereferencing operators are:

(i) ::* This declare a pointer to a member of class,

For example: `void (T::*P1) (int) = T::f1;`

Here, class is T. The above definition says that the member function to which P1 points will return nothing but requires int argument. It also initialise the pointer P1 to the address of the function f1 of class T.

(ii) .* To access a member using object name and a pointer to that member

For example:

```
T   01;
(01 .*P) (3)  //call function through pointer
```

(iii) →* To access a member using a pointer to the object and a pointer to that member.

For example:

```
T   01;
01  *P;
P  → *f1(2);
```

3. Memory Management Operator:

- For dynamic memory allocation, C++ provides new operator. In C, we have malloc( ) and calloc( ) functions for dynamic allocation. For dynamic deallocation, C++ provides delete( ) operator, which is same as free( ) function in C.

(i) **new operator:** The new operator is used to allocates the memory.

Syntax: `ptr_var = new datatype;)`

For example:

```
char *P;
P = new char[5];    //reserves 5 bytes of contiguous memory of
                     characters
int *x;
x = new int;        //allocate 2 bytes for integer
```

(ii) **delete operator:** The delete operator deallocates or frees the memory allocated by new.

Syntax: `delete ptr_var;`

The ptr_var is the pointer to a data object which is created by new.

For example,

```
delete P;
delete x;
```

If the array is allocated by new, then to free it we use the following syntax,

```
delete [size] ptr_var;
```

If the size is not specified then entire array get deleted.

For example, `delete [ ] a;`

Program 3.4: Program for new and delete operators.

```
#include<iostream>
#include<string>
using namespace std;
void main()
{
   char *x;
   x = new char[10];
   cout<<"Enter String";
   cin>>x;
   cout<<"\n The string is:"<<x<<endl;
   delete x;
}
```

Output:

```
The string is: COMPUTER
```

4. Insertion and Extraction Operators:

- In C++, cin and cout operators are used for input and output operations respectively. C++ language I/O operators can be used by including 'iostream' header file in the program.
- **Extraction operator** (>>) is used to read data from a standard input device to the memory items.
- **Insertion operator** (<<) is used to write data from memory to a standard output device.
- In C++, the input stream uses 'cin' object to read data and the output stream uses 'cout' object to display data on the screen. The cin and cout are pre-defined streams for input and output of data.
- The data type is identified by these functions using operator overloading of the operators << (insertion operator) and >> (extraction operator).

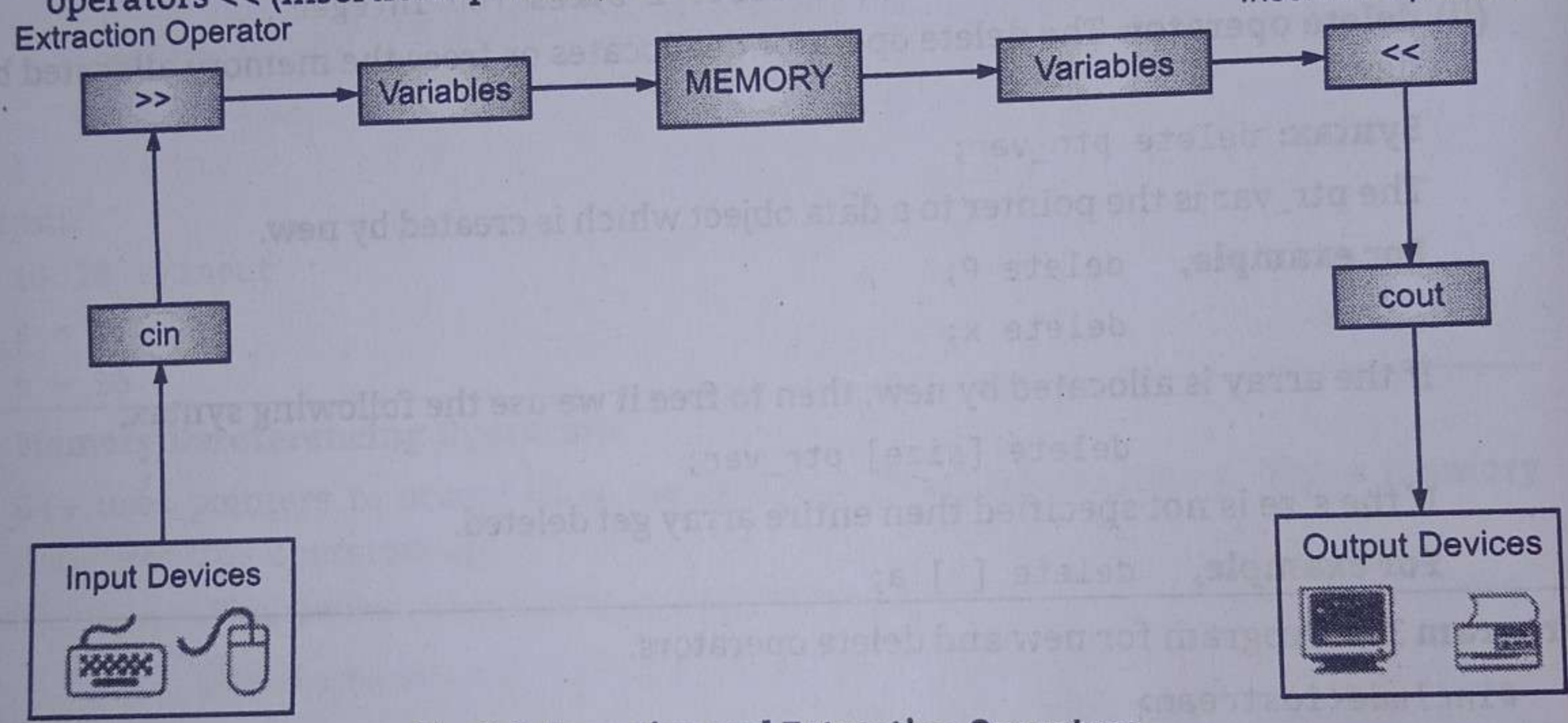


Fig. 3.2: Insertion and Extraction Operators

- The >> (extraction) operator in C++ is used with the cin statement. The operator >> also called as get from operator.
- **For example:**

```
int x;
cin>>x;
```

- In above example, the >> (extraction) operator extracts a value such as 40 from the keyboard and assigns it to the variable x, as shown in Fig. 3.3.

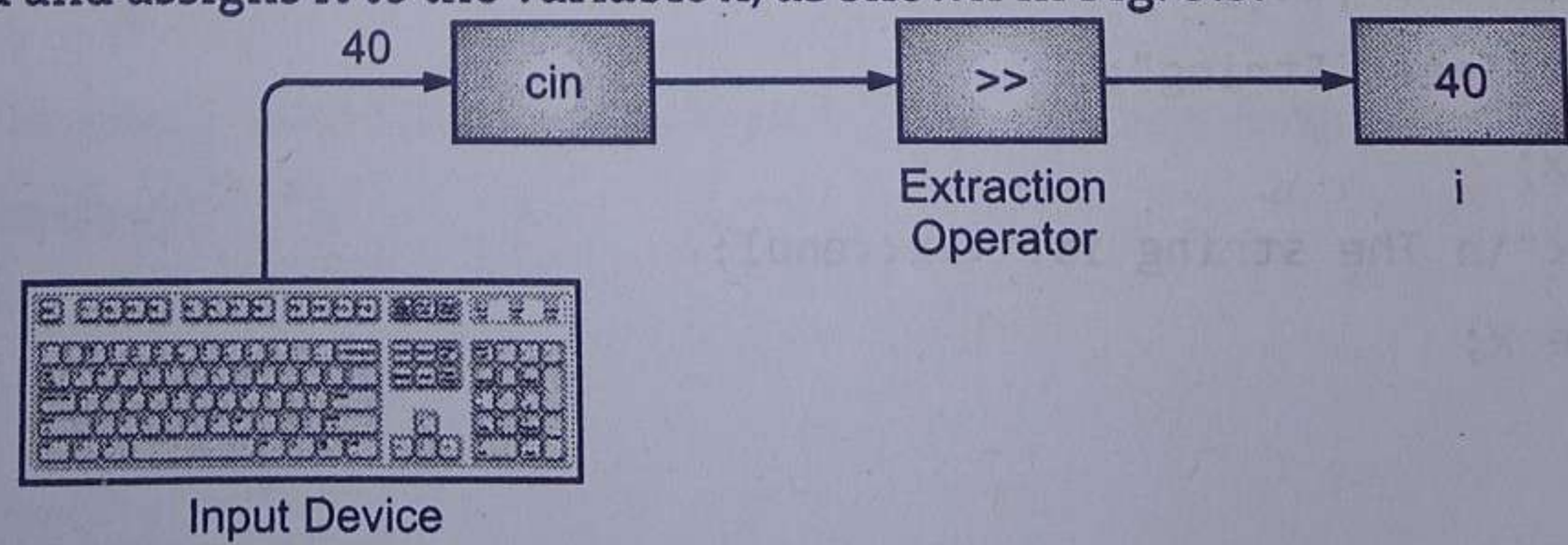


Fig. 3.3: Extraction Operator

- The << (insertion) operator in C++ is used with the cout statement, which is used to send the result to an output device (monitor).
- The << operator is used to insert the contents of a variable on its right to the object on its left. The operator << also called as put to operator.
- **For example:**

```
int avg = 10;
cout<<"Average is: "<<avg<<endl;
```

- In the above example, the << operator first sends the string Average is: to cout. Next, the content of the variable avg is sent to cout and finally the keyword endl is sent, which transfers the control to the end of the line, as shown in Fig. 3.4.

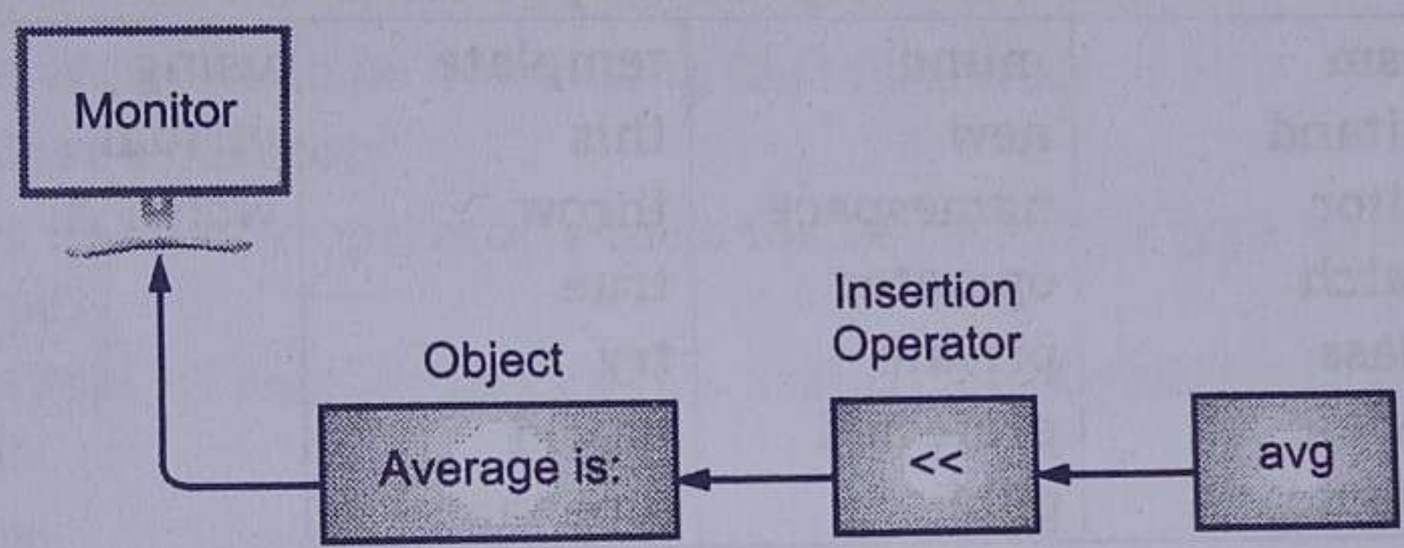


Fig. 3.4: Insertion Operator

- The << and >> operators for shifting are overloaded in ostream and istream class respectively.
- The general **syntax** for reading the data from the keyboard is:

```
cin >> var1, >> var2 >> ..... >> varn;
```

  Where, var1 and var2 are any, C++ predefined variables.
- This statement in the program will make the compiler to wait for the input data from the keyboard.
- The operator >> reads the data character by character and assigns it to the indicated location.
- The reading for the variable will be terminated at the encounter of a white space or a character that does not match with destination type.
- The general **syntax** for displaying data on screen is:

```
cout << data1 << data2 << ..... <<datan;
```

3.4 KEYWORDS

- The keyword implements specific C++ language features.
- Keywords are explicitly reserved identifiers and cannot be used as name for the program variables.
- It is mandatory that all the keywords should be in lowercase letters.
- The keywords are reserved words, predefined by the language and user cannot change.
- Keywords are used for specific purposes in C++ and compiler can interprets these words.

- Table 3.6 shows the list of keywords which are common in C and C++ language and Table 3.7 shows the keywords specific to C++.

Table 3.6: Keywords common to C and C++

auto	default	float	return	union
break	do	for	shortsigned	unsigned
case	double	goto	sizeof	void
char	else	ifint	staticstruct	volatile
const	enum	long	switch	while
continue	extern	register	typedef	

Table 3.7: Keywords specific to C++

asm	inline	template	using
bitand	new	this	virtual
bitor	namespace	throw	wchar_t
catch	operator	true	
class	private	try	
delete	protected	typeid	
friend	public	typename	

New Keywords in C++:

C++ uses some additional keywords which are supporting new features in C++.

Table 3.8: C++ new keywords

bool	false	true
const_cast	mutable	typeid
dynamic_cast	namespace	typename
explicit	reinterpret_cast	using
export	static_cast	wchar_t

3.5 TYPE CASTING IN C++

- In C++, there are two types of type casting i.e. implicit type casting & explicit type casting.

1. Implicit Type Conversion:

- This is also known as automatic **type conversion**. It is done by the compiler on its own. There is no external trigger required by the user to typecast a variable from one type to another.
- This occurs when an expression contains variables of more than one type. So, in those scenarios automatic type conversion takes place to avoid loss of data. In automatic type conversion, all the data types present in the expression are converted to data type of the variable with the largest data type.
- For example:

```
short a=2000;
int b;
b=a;
```

- Here, the value of a has been promoted from short to int and we have not had to specify any typecasting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.

2. Explicit Type Conversion:

- Explicit type conversion is also known as **type casting.** It is user defined type conversion. In explicit type conversion, the user converts one type of variable to another type.
- Explicit type conversion can be done using Cast operators.

Conversion using Cast Operator:

- Cast operator is an unary operator which forces one data type to be converted into another data type.
- There are four types of casting in C++, i.e. Static Cast, Dynamic Cast, Const Cast and Reinterpret Cast.

1. Typeid Operator:

The typeid operator returns an identifier for an object, which can be compared to other type and printed appropriate message. We have to include the header file<typeinfo.h> to use the results of the operator.

```
#include<typeinfo.h>
```

We can use typied for classes and objects both.

For example: The function test the condition that an object is of type x.

```
if (typeid(object) == typeid(x))
cout<<"object is x";
```

The typeid operator returns an object of class type_info, which has a member function called name. This function returns a null terminated character string.

```
cout<<(typeid(objectname).name());
```

2. static_cast operator:

It is used for type casting, (any type conversion). It also allow to cast a pointer of a derived class to its base class. The syntax is,

```
static_cast <data type>  (object name)
```

For example:

1.
```
int a = 10;
float b = static_cast <float> a;
```

2.
```
class x
{
    ........
};
```

```
class y: public x
{
    ........
};
x * P1 = new x;
y * P2 = static_cast <y*> P1;
```

3. dynamic_cast operator:

It is used type casting same as static_cast operator but at runtime. It is used with pointers and references to objects. The syntax is:

```
dynamic_cast <data type> (object name)
```

Here objectname is base class object.

Example:

```
void f1(employee *P)
{
    manager *m;
    m = dynamic_cast<manager*>(P);
    if (m)
    {
        ............
        ............
    }
}
```

4. cons_cast operator:

This type of casting manipulates the const attribute of the passed object. It is either override or removed.

For example:

```
class x
{ ......... }
const x *P1 = new x;
x *P2 = const_cast <x*> (P1);
```

5. reinterpret_cast operator:

This is used for pointer casting i.e. a pointer type change to any other type.

For example:

```
int i;
float f;
int * P1;
float * P2;
P1 = reinterpret_cast<int*>i;
P2 = reinterpret_cast<float*>f;
```

3.6 REFERENCE VARIABLE [W-18]

- A reference variable is an alias or an alternative name for a previously defined variable.

For example:

```
int a = 20;
int &b = a;          // & is a reference operator
2132   ← Address
[20]   ← Value
a      ← Variable name
b      ← reference to a
```

- Manipulating a reference to an object allows manipulation of the object itself. This is because no separate memory is allocated for a reference.
- A reference to a variable is created using the address of operator (&). When the & operator is used in the declaration, it becomes the reference operator.
- A reference variable is declared as follows:

```
datatype & ref_var_name = variable_name;
```

Where, ref_var_name is name of reference variable.

- In above example, a is an integer variable whose address (*l*-value) is 2132 which is intialize to 20. In the second statement, 'b' is a reference to 'a'. If we manipulate the value of b (say 10), then the value of the memory location 2132 gets changed or in other words the value of 'a' changes.

Program 3.5: Program for reference variable.

```
#include<iostream>
using namespace std;
void main()
{
    int p = 50;
    int *a = &p;
    int &b = *a;
    cout<<"*a="<<*a<<"b="<<b<<endl;
    int x = 10;
    int &y = x;
    x = x + 10;
    cout<<"x="<<x<<"y="<<y<<endl;
}
```

Output:

```
*a = 50    b = 50
x = 20     y = 20
```

Reference and Function Call:

- We can use reference variable in passing arguments to functions. When we use pass by reference mechanism, the function receives the *l*-value of the argument rather than the copy of the passed argument.

Program 3.6: Use of reference in function arguments.

```
#include<iostream>
using namespace std;
void swap (int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
int main()
{
    int x=10,y=20;
    swap (x, y);
    cout<<"x="<<x<<"y="<<y;
}
```

Output:

```
x = 20    y = 10
```

- Here, x and y are alias of a and b respectively. So one location we have two names as shown in below:

before call **at call**

x [10] a [10] ← x is referring same location as a

y [20] b [20] ← y

after execution

[10] ←y

[20] ←x

3.7 CLASSES AND ACCESS SPECIFIERS [W-18, S-19]

- Access specifiers are used to restrict (limits) the accessibility of class members. Access specifiers defines how the members of the class can be accessed.
- Each member of a class is associated with an access specifiers. The access specifiers of a member control its accessibility as well as determines the part of the program that can directly access the member of the class.
- In the class declaration syntax, the keywords private, public and protected are known as access specifiers (also known as visibility modes).

- An access specifier specifies the access rules for members following it until the end of the class or until another access specifier is encountered.
- When a member is declared private, it can be accessed only inside the class while a public member is accessible both inside and outside the class. Protected members are accessible both inside the class in which they are declared as well as inside the derived classes of the same class.
- Once, an access specifier has been used, it remains in effect until another access specifier is encountered or the end of the class declaration is reached. An access specifier is provided by writing the appropriate keyword (private, public or protected) followed by a colon ' : '.
- Note that, the default access specifier of the members of a class is private. i.e., if no access specifier is provided in the class definition, the access specifier is considered to be private.
- In C++, we can set the access of a class's members using the access specifiers. We can make members private (visible only to code in the same class), protected (visible only inside the same class and classes derived from it), or public (visible to code inside and outside the class). By default, all the class has private access by default.
- The general **syntax** of a class declaration that uses the private and public access specifiers are given below:

```
class ClassName
{
    private:
        //Declarations of private
        //members appear here ...
    public:
        //Declarations of public
        //members appear here ...
};
```

- For example:

```
class book
{
    //private by default
    char title [30] ; //variables declaration
    float price;
    public:
        void getdata(char [],float); //function declaration
        void putdata ();
} ;
```

- In above example, a class name book with two data members title and price and two member functions getdata () and putdata () is created. As no access specifier is provided for data members, they are private by default, whereas, the member

functions are declared as public. It implies that the data members are accessible only through the member functions while the member functions can be accessed anywhere in the program.

3.8 DEFINING DATA MEMBERS AND MEMBER FUNCTIONS

- All the variables inside the class are called data members. These variables are of any basic data type, derived data type, user defined data type or objects of other class.
- Any function declare inside a class is called a member function of that class. Only the member functions can have access to the private data members and private functions. By default all members of class are private.
- The general **syntax** of a class declaration is shown below:

```
class class_name
{
    private:
        variable declarations;
        function declarations;
    public:
        variable declarations;
        function declarations;
};
```

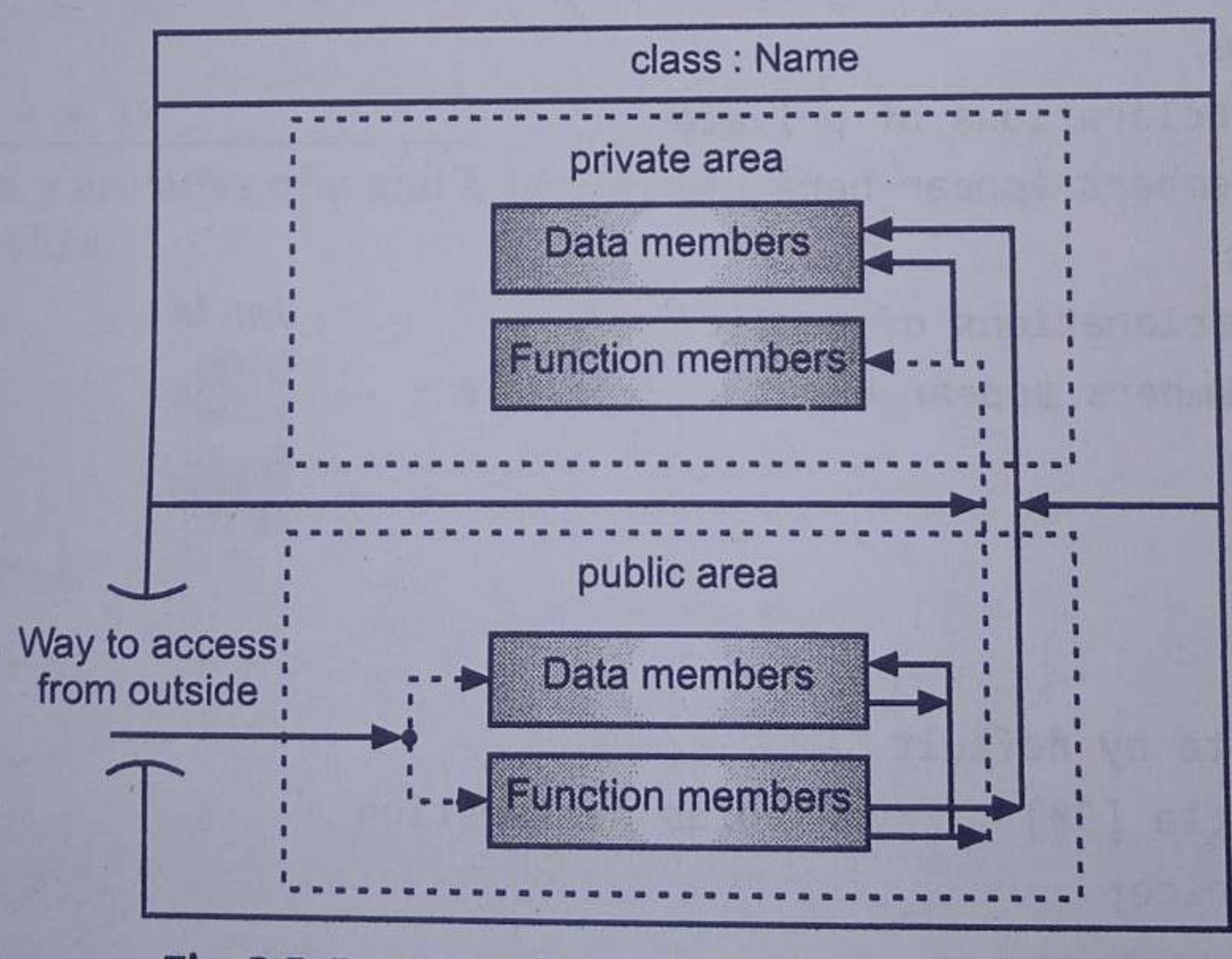


Fig. 3.5: Data Member and Member Function

- Member functions of a class are defined in following two ways: [S-19]
  1. Inside the class definition.
  2. Outside the class.
- In both ways, only syntax of definition of member function changes but body of function (code) is remain same.

1. Inside the Class Definition:

- Functions which are defined inside the class are similar to normal functions and they are handle automatically and inline functions. Normally, function which are small are define inside the class:

Example of student class:

```
class student
{
    int roll;
    char name[20];
    public:
        void getdata()          //definition of member function
        {
            cin>>roll>>name;
        }
        void putdata()          //definition of member function
        {
            cout<<roll<<name;
        }
};
```

2. Outside the Class:

- The functions are define after the class declaration, however the prototype of function should be appear inside the class i.e. declaration only (not definition). Since, the function is defined outside the class, there should be some mechanism to know the identity of function to which class they belong we have to use scope resolution operator (::) to identify the function belongs to which class.
- We can also have a function with the same name and same argument list in different classes, since the scope resolution operator will resolve the scope of the function.
- The general **syntax** is:

```
return type class_name :: function_name (argument list if any)
{
    ................ //body of function
}
```

Example:

```
class student
{
    ................
    ................
    public:
        void getdata();  //declaration of member function
        ................
        ................
};
```

```
void student::getdata()
{
    ...............
}
```

Program 3.7: Program for class declaration and creation of objects.

```
#include<iostream>
using namespace std;
class date
{
   private:
      int d;
      int m;
      int y;
   public:
      void get (int Day, int Month, int Year);
      void display();  //declaration
};
void date::get (int Day,int Month,int Year)          //definition
{
    d=Day;
    m=Month;
    y=Year;
}
void date::display()   //definition
{
    cout<<d<<"-"<<m<<"-"<<y<<endl;
}
int main()
{
    date d1, d2, d3;        //date objects d1, d2 & d3
    d1.get (26, 3, 1968);  //call member function
    d2.get(15, 9, 1978);
    d3.get(12, 3, 1992);
    cout<<"Birth Date of the first child:";
    d1.display();
```

```
    cout<<"Birth Date of the second child:";
    d2.display();
    cout<<"Birth Date of the third child:";
    d3.display();
}
```

Output:

```
Birth Date of the first child  : 26 - 3 - 1968
Birth Date of the second child : 15 - 9 - 1978
Birth Date of the third child  : 12 - 3 - 1992
```

Note:

- A member function can call another member function directly.
- No memory is allocated till the point you defined objects of that class.

3.9 ARRAYS AND ARRAY OF OBJECTS [W-18]

- The array of variables of class data type is called Array of Objects.
- An array of objects is an array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built-in data type.
- The **syntax** for declaring an array of objects is as follows:

```
class_name array_name[size];
```

- **Example:**

```
class student
{
    int roll;
    char name[20];
    public:
        void getdata();
        void putdata();
};
```

If we create object of class student,

```
student s;    // single object
```

The array of objects is created as:

```
student s[100];      //array of student
```

Here, the array of s contains 100 objects.

We can also have the array of different categories of the student,

```
student      sc[10];
student      open[30];
student      obc[20];
```

- The array of **sc** contains 10 objects, the array of open contains 30 objects, and the array of **obc** contains 20 objects. The behaviour and attributes of each object is define is class where the identifier name is same but values are different.

  For example, Identifier roll has values 100, 101 .....
- The array of object is stored in memory as same as how multidimensional array are stored. The space is required only for data item of the class. The array of student is representing in Fig. 3.6.
- We have discussed how single object access the data items on member function. In same way, using array of objects we can access members of class (using dot).

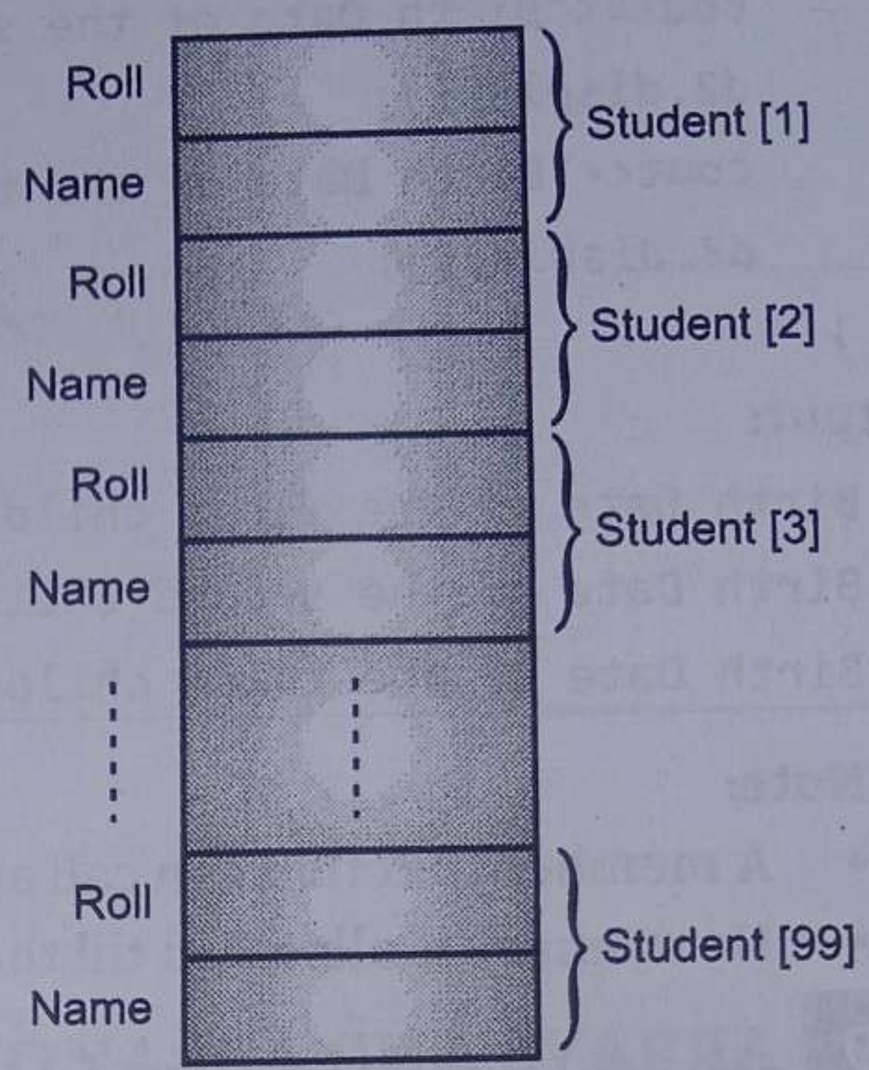


Fig. 3.6: Storage for Data Items in Array of Objects

For example, `student[i].getdata();`

Here, we can input the ith element data of student array.

For example, `student [i].putdata();`

Hence, we can display the ith element of the student array.

Program 3.8: Program for array of objects. [S-18, 19]

```
#include<iostream>
using namespace std;
class student
{
    int roll;
    char name[20];
    public:
        void getdata (void);
        void putdata (void);
};
void student:: getdata(void)
{
    cout<<"Enter roll:";
    cin>>roll;
    cout<<"Enter name:";
    cin>>name;
}
```

```
void student::putdata(void)
{
    cout<<"roll:"<<roll<<endl;
    cout<<"name:"<<name<<"\n";
}
int main()
{
    int i;
    student s[3]; //array of size three
    for(i=0;i<3;i++)
    {
        cout<<i+1<<"Student Information"<<endl;
        s[i].getdata();
    }
    for (i=0;i<3;i++)
    {
        s[i].putdata();
    }
    return 0;
}
```

Output:

```
1 Student Information
    Enter roll: 200
    Enter name: Nirali
2 Student Information
    Enter roll: 201
    Enter name: Prajakta
3 Student Information
    Enter roll: 202
    Enter name: Anita
    roll: 200
    name: Nirali
    roll: 201
    name: Prajakta
    name: 202
    name: Anita
```

3.10 USAGE OF NAMESPACE

- Consider a situation, when we have two persons with the same name, Seema, in the same class. Whenever, we need to differentiate them definitely we would have to use some additional information along with their name, like either the area, if they live in different area or their mother's or father's name, etc.

- Same situation can arise in the C++ applications. For example, we might be writing some code that has a function called abc() and there is another library available which is also having same function abc(). Now the compiler has no way of knowing which version of abc() function we are referring to within the code.
- A namespace is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, we can define the context in which names are defined. In essence, a namespace defines a scope.
- A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc.) inside it. Namespaces are used to group related names and to avoid clashes between similar names in different parts of large programs.

Defining a Namespace:

- A namespace definition begins with the keyword 'namespace' followed by the namespace name as follows:

```
namespace namespace_name
{
    // code declarations ........
}
```

- To call the namespace enabled version of either function or variable we use,

```
name::code;  // code could be variable or function
```

Program 3.9: Program for namespace concept.

```
#include <iostream>
using namespace std;
// first name space
namespace space_one
{
    void func()
    {
        cout << "Inside space_one";
    }
}
// second name space
namespace space_two
{
    void func()
    {
        cout << "Inside space_two";
    }
}
```

```
int main ()
{
    // Calls function from first name space.
    space_one::func();
    // Calls function from second name space.
    space_two::func();
    return 0;
}
```

Output:

```
Inside space_one Inside space_two
```

3.11 MANAGING CONSOLE I/O

- C++ defines its own object oriented I/O system; it has its own object oriented way of handling data input and output. These Input/Output operations are carried out using different stream classes and their related functions.
- Until now we have used the object of the istream class (cin) and object of ostream class (cout); with operators >> and << respectively. For formatting purpose we use different streams and their functions in C++.

3.11.1 C++ Streams

- A stream is sequence of bytes.
- Stream acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent.
- In C++, to manage the data flow we have different stream classes, Fig. 3.7 shows stream classes for console I/O.

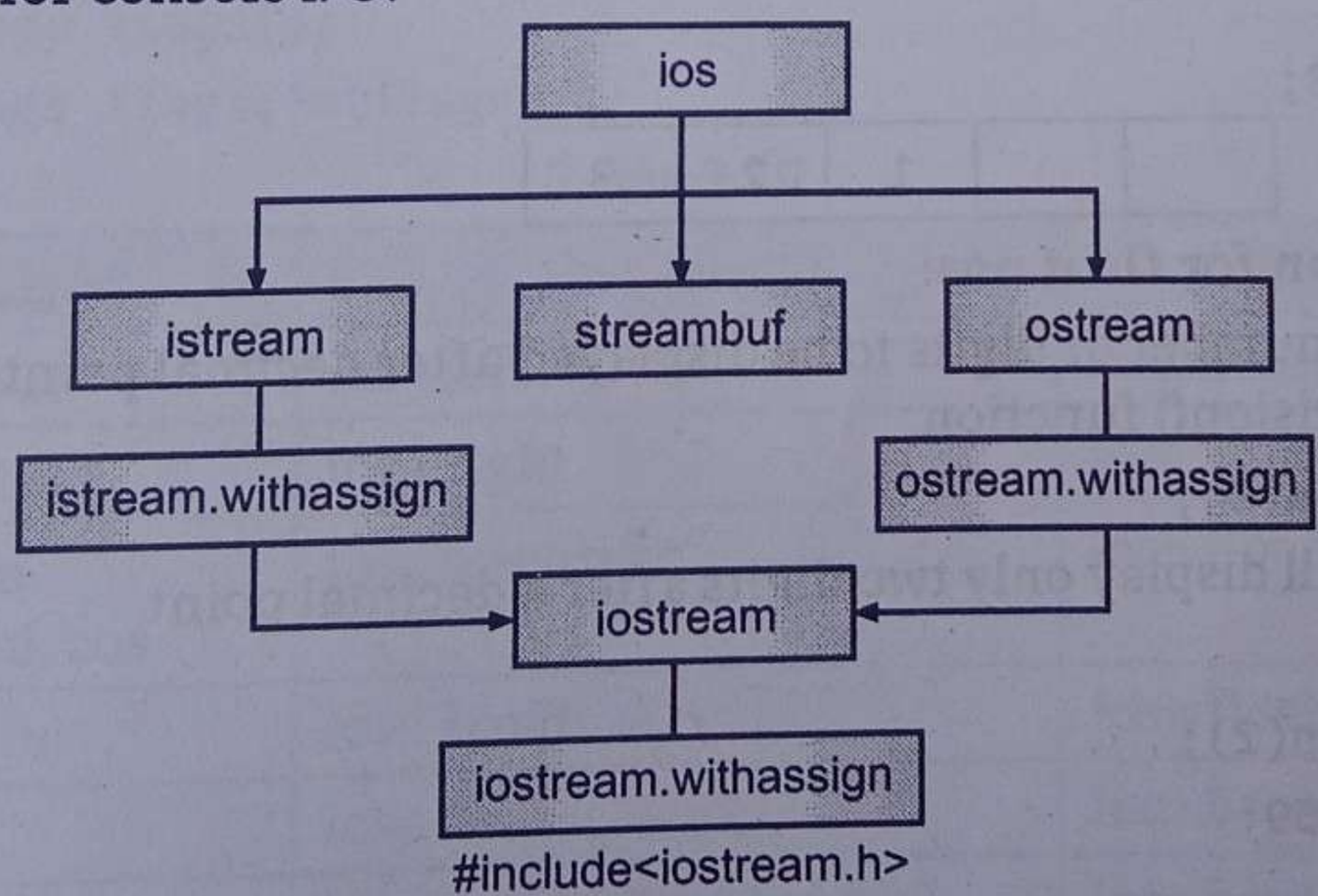


Fig. 3.7: Stream class for console I/O

1. **istream class:** It is a derived class of ios and hence, inherits the properties of ios. It defines input functions such as get( ), getline( ) and read( ). In addition, it has an overloaded member function, stream extraction operator >>, to read data from a standard input device to the memory items.

2. **ostream class:** It is a derived class of `ios` and hence inherits the properties of `ios`. It defines output functions such as `put( )` and `write( )`. In addition, it has an overloaded member function, stream insertion operator <<, to write data from memory to a standard output device. `cerr` object is of ostream class used for displaying error messages.
3. **iostream class:** It is derived from multiple base classes, istream and ostream, which are inturn inherited from class `ios`. It supports both input and output stream operations.

- The classes' istream_withassign, ostream_withassign and iostream_withassign add the assignment operators to their parent classes.

3.12 USAGE OF MANIPULATORS [S-18, 19]

- Manipulators are the functions used in Input/Output statement.
- All the manipulators are defined in header field **iomanip.h.**
- We can use chain of manipulator that is one or more manipulators in statement.

Table 3.9: Manipulators

Manipulators	Equivalent I/O function
setw()	width()
setprecision()	precision()
setfill()	fill()
setioflags()	setf()
resetioflags()	unsetf()

1. Setting of field width:

- setw() is used to define the width of field necessary for the output for a variable.
- For example: `setw(5);` This function will reserve 5 digits for a number.

```
setw(5);
cout << 123;
```

Output:

		1	2	3

2. Setting precision for float nos:

- We can control number of digits to be displayed after decimal point for float numbers by using setprecision() function.

```
setprecision(2);
```

- This function will display only two digits after a decimal point.
- For example:

```
setprecision(2);
cout<<3.14159;
```

Output: `3.14`

3. Filling of unused positions:

- Filling of unused positions of the field can be done by using setfill().

```
setfill('*');
```

- Here, unused positions will be filled by using * sign.

```
setw(5);
setfill ('$');
cout << 123;
```

Output:

$	$	1	2	3

4. setbase():

- It is used to show the base of a number, for example:

```
setbase(10);
```

- It shows that all numbers are decimal numbers.

1---------1
2---------4
3---------9
4---------16

Let justify
flags should be set
setf (arg 1, arg 2);
from where which type of flat

5. Flags:

- C++ defines some format flags for standard input and output, which can be manipulated with the `flags()`, `setf()` and `unsetf()` functions.
- The flags() function either return the format flags for the current stream or sets the flags for the current stream to be f.

Syntax: `fmtflags flags();`
`fmtflags flags(fmtflags f);`

Table 3.10

Format required	flag (arg 1)	Bit field (arg 2)
Left justified output	ios:: left	ios:: adjustfield
Right justified output	ios:: right	ios:: adjustfield
Scientific notation trigonometric – sin, cos	ios:: scientific	ios:: floatfield
Fixed point notation	ios:: fixed	ios:: floatfield
decimal base	ios:: dec	ios:: basefield
octal base	ios:: oct	ios:: basefield
hexadecimal base	ios:: hex	ios:: basefield

6. setf() and unsetf():

- setf() function can also be used with only one argument. Arguments will be as follows:

Table 3.11

arg1	Use
ios:: showbase	It helps to use base indicator an output
ios:: uppercase	It helps to use uppercase letters.
ios:: skipws	It skips (blank) white spaces from input
ios:: showpoint	It helps to display trailing zeros.
ios:: showpos	It helps to display + sign for number.

- The function unsetf( ) is used to clear the given flags associated with the current system.

  Syntax: `void unsetf(fmtflags flags);`

7. **endl:**

- This manipulator used in an output statement, causes a linefeed to be inserted. It has the same effect as using the new line character "\n".

  For example:

```
cout <<"a=" << a << endl
     <<"b=" << b << endl
     <<"c=" << c << endl;
```

If we assume value of variable as 2215, 181 and 17 the output will:

a = | 2 | 2 | 1 | 5 |

b = | 1 | 8 | 1 |

c = | 1 | 7 |

Program 3.10: Program to display students data.

```
#include<iostream>
using namespace std;
class student
{
   int marks[5];
   int total=0;
   char name[20];
  public:
   void read_data();
   void display_data();
};
void student:: read_data()
{
   int i;
   cout <<"Enter student name:";
   cin >> name;
   cout <<"Enter 5 subject marks:";
```

```
    for (i = 0;  i < 5;  i++)
    {
        cin >> marks[i];
        total = total + marks[i];
    }
}
void student:: display_data()
{
    cout <<"\n Name:"<< name;
    cout <<"\n Marks in 5 subjects:";
    for (int i = 0; i < 5; i++)
    cout <<"\t" << marks[i];
    cout <<"Total ="<< total;
}
int main()
{
    student s;
    s.read_data();
    s.display_data();
    return(0);
}
```

Output:

```
Enter student name:Rekha
Enter 5 subject marks:20
50
30
60
40
Name:Rekha
Marks in 5 subjects:   20   50   30   60   40   Total = 200
```

Program 3.11: Program to declare class account having data members principle, rate_of_interest, no_of_years. Accept this data for one object and find out simple interest.

```
#include<iostream>
using namespace std;
class account
{
    private:
        int p, n, r;
        float si;
```

```
    public:
        void getdata()
        {
            cout<<"Enter Principle, Rate_of_interest and number_of_years:\n";
            cin>> p >> r >> n;
        }
        void calculate ()
        {
            si = (p * n * r) / 100;
        }
        void display()
        {
            cout <<"Simple Interest =";
            cout << si;
        }
};
int main()
{
    account a;
    a.getdata ();
    a.calculate();
    a.display();
    return(0);
}
```

Output:

```
Enter Principle, Rate_of_interest and number_of_years:
5400 12 5
Simple Interest = 3240
```

Program 3.12: Program to declare a class rectangle having data members length and breadth. Accept this data for one object and display area and perimeter of rectangle.

```
#include<iostream>
using namespace std;
class rectangle
{
    int len, bred, area, peri;
public:
    void accept( );
    void calculate( );
    void display( );
};
```

```
void rectangle:: accept()
{
   cout <<"Enter length and breadth:";
   cin >> len >> bred;
}
void rectangle:: calculate()
{
   area = len * bred;
   peri = 2 * (len + bred);
}
void rectangle:: display()
{
   cout<<"Area of rectangle:" << area;
   cout<<"\n Perimeter of Rectangle:"<<peri;
}
int main( )
{
   rectangle r1;
   r1.accept( );
   r1.calculate( );
   r1.display( );
   return(0);
}
```

Output:

```
Enter length and breadth:9 5
Area of rectangle:45
Perimeter of Rectangle:28
```

Program 3.13: Program to declare a class book having data members as title and author. Accept this data for five books and display this accepted data.

```
#include<iostream>
using namespace std;
class book
{
   private:
      char auth[20];
      char title [50];
   public:
      void accept( );
      void display( );
} b[2];
```

```
void book:: accept( )
{
    cout <<"Enter Title of the book:"<<endl;
    cin >> title;
    cout <<"\nEnter author name:"<<endl;
    cin >> auth;
}
void book:: display( )
{
    cout << "Book Name:" << title;
    cout << "\nAuthor:" << auth;
}
int main()
{
    int i;
    for (i = 0;  i < 5;  i++)
    {
        b[i] .accept( );
        b[i] .display( );
    }
    return(0);
}
```

Output:

```
Enter Title of the book:C++ programming
Enter author name:  Bharambe Manisha
Book Name: C++ programming
Author: Bharambe Manisha
Enter Title of the book: Java programming
Enter author name: Balguruswamy
Book Name: Java programming
Author: Balguruswamy
```

Program 3.14: Program to display book information using class object.

```
#include<iostream>
using namespace std;
class book
{
    int bookno;
    char bookname[20];
    char author[20];
    float price;
    public:
        void getdata();
        void display()
};
void book::getdata()
{
    cout<<"enter book number";
    cin>>bookno;
```

```
    cout<<"enter book name";
    cin>>bookname;
    cout<<"enter author name";
    cin>>author;
    cout<<"enter book price";
    cin>>price;
}
void book::display()
{
    cout<<"book number"<<bookno<<endl;
    cout<<"book name="<<bookname<<endl;
    cout<<"author="<<author<<endl;
    cout<<"price="<<price<<"Rs.";
}
int main()
{
    book b;
    b.getdata();
    b.display();
}
```

Output:

```
Book no = 2503
Book name = C++
Author = Bharambe
Price = 99.99 Rs.
```

Summary

- C++ provides various types of tokens that include keywords, identifiers, constants, strings, and operators.
- The C++ keywords are reserved words by the compiler. All C language keywords are valid in C++ and few additional keywords are added by ANSI committee.
- A reference variable provides an alternative name (alias) for a previously defined variable. Both the variables refer to the same data object in the memory. Hence, change in the value of one will also be reflected in the value or the other variable.
- A reference variable must be initialized at the time of declaration, which establishes the correspondence between the reference and the data object that it names.
- A major application of the scope resolution (::) operator is in the classes to identify the class to which a member function belongs.
- An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide various types of operators like Arithmetic operators (+, -, *, /* etc.), Relational operators (<, >, != etc.), Logical operators (&&,||, ! etc.), Bitwise operators (^, &, |, >>, << etc.), Assignment operators(=, +=, *=, /= etc.) and so on.
- A manipulator changes the stream or state of the stream passed to it as an argument.
- C++ programming environments predefine a bunch of manipulators in the header files iostream.h and iomanip.h.
- In C++, endl manipulator is used to inserts a newline character in the output stream and then flushes the stream.

- In C++, setprecision manipulator is used to sets the floating-point precision to be the number of digits specified by the argument.
- In C++, setfill manipulator used to sets the fill character to be the character specified by the argument.
- In C++, setw manipulator used to sets the field width to the value specified by the argument.

Check Your Understanding

1. The _____ operator extracts the value of a variable from cin object.
   (a) extraction (b) insertion
   (c) instance (d) none
2. The words which has predefined meaning and cannot be changed by the users are known as _____.
   (a) constants (b) identifiers
   (c) keywords (d) none
3. Which of the following is not keyword?
   (a) bool (b) abstract
   (c) protected (d) mutable
4. Which of the following keywords is used to control access to a class member?
   (a) default (b) break
   (c) protected (d) goto
5. The dynamic memory allocation can be done through _____ operator.
   (a) new (b) delete
   (c) pointer (d) none
6. What is ?: called?
   (a) ternary operators (b) binary operator
   (c) arithmetic operator (d) none of these
7. Which keyword is used to access to variable in namespace?
   (a) static (b) using
   (c) const (d) dynamic
8. _____ is default namespace defined by C++.
   (a) using (b) include
   (c) std (d) none of these

Answers

1. (b)	2. (c)	3. (b)	4. (c)	5. (a)	6. (a)	7. (b)	8. (c)

Practice Questions

Q.I Answer the following questions in short:

1. What is meant by >> and << operators.
2. Enlist data types in C++.
3. Give the syntax to create a reference variable.
4. What is the purpose of scope resolution operator.
5. Give two methods which are used to define member function.

Q.II Answer the following questions:

1. What are the applications of void data type in C++?
2. What is the application of the scope resolution operator:: in C++?

3. What are the advantages of using new operator as compared to the function malloc()?
4. Explain keywords in C++.
5. Explain working of extraction and insertion operators.
6. What is manipulator? Enlist them with example.
7. Explain the working of << and >> operators.
8. What is memory management operator? Explain the operator 'new' with example. Give any two advantages of 'new' over 'malloc'.
9. Write a C++ program to accept student information as sno, sname, sub1 and sub2 for five students using array of objects. Calculate total marks and display output.

Q.III Define terms:

1. Reference variable.
2. Implicit casting
3. Explicit casting
4. Type casting

Previous Exam Questions

Summer 2018

1. The words which has predefined meaning and cannot be changed by the users are known as _____. **[1 M]**
   (i) constants
   (ii) identifiers
   (iii) keywords
   (iv) none

Ans. (iii) keyword

2. Write C++ program to accept employee details like empno, empname, city, idate salary, deptname using accept() and display() function for array of object. **[5 M]**

Ans. Refer to Program 3.8.

3. Explain the manipulators uses in C++. **[4 M]**

Ans. Refer to Section 3.12.

4. Explain scope resolution operator with example. **[4 M]**

Ans. Refer to Section 3.3.1.

Winter 2018

1. _____ header file is used for manipulators. **[1 M]**
   (a) <iomanipulator.h>
   (b) <manip.h>
   (c) <stdmanip.h>
   (d) <iomanip.h>

Ans. (d) <iomanip.h>

2. What is the use of : : operator? **[1 M]**

Ans. Refer to Section 3.3.1.

3. Illustrate array of objects with examples. **[5 M]**

Ans. Refer to Section 3.9.

4. Give the general format of class and state the significance of private, public and protected access specifier. [4 M]

Ans. Refer to Section 3.7.

5. What is the purpose of reference variable? Explain with the help of a suitable example. [4 M]

Ans. Refer to Section 3.6.

Summer 2019

1. Scope resolution operator is used to unhide _____ variables. [1 M]
   (a) Global
   (b) Local
   (c) Main
   (d) None of the above

Ans. (a) Global

2. The members declared in private section can be accessed by _____. [1 M]
   (a) Member functions of same class
   (b) Friend function of class
   (c) Derived classes
   (d) Member functions of their same class and friend of this class

Ans. (a) Member functions of same class.

3. Write a C++ program to define class books having members title and price. Define member functions accept() and display(). Create 5 objects of class accept and display the info of books using array of objects. [5 M]

Ans. Refer to Program 3.8.

4. Explain the ways to defining member functions. [4 M]

Ans. Refer to Section 3.8.

5. Give purpose of protected keyword. [3 M]

Ans. Refer to Section 3.7.

6. Define manipulator. Explain setfill( ) with example. [3 M]

Ans. Refer to Section 3.12.

4... Constructors, Destructors, Function and Operator Overloading

Objectives...

- To study Concepts of Constructors and Types of Constructors.
- To understand multiple constructor in a class.
- To study Dynamic Constructor.
- To understand the Concepts of Destructors.

4.1 INTRODUCTION

- C++ provides a special member function called as Constructor.
- A constructor is an overloaded user defined function which enables an object to initialize itself when it is created. This is known as automatic initialization of objects.
- The main task of constructor is to initialize the objects of its class.
- The complement of the constructor is the destructor.
- A complementary user defined member function is applied automatically to each class object after the last use of that object is known as Destructor.
- Destructors are used to destroy an object.
- Constructor is automatically called when object is created.
- Constructors are also called when an object is created as part of another object.

4.2 FUNCTIONS

- Functions are the basic building blocks of C++ and the place where all program activity occurs. A function is a group of statements that together perform a task.
- A function is a subprogram that acts on the program data and often returns values.
- A program written with numerous functions is easier to maintain, update and debug than one very long or large program.

- Functions are used to implements large programs. In structured programming functions are used to break the larger programs into smaller ones; which makes debugging of program easier.
- Each function has its own name. When that name is encountered in a program, the execution of the program branches to the body of that function.

Features of Functions:

1. The structure of a function definition is look like the structure of main(), with its own list of variables declaration and program statements.
2. A function can have a list of zero or more parameters inside its brackets, each of which has a separate type.
3. A function may have more than one "return" statement in which case the function definition will end execution as soon as the first "return" is reached.
4. Function declarations are like variable declaration, (functions specify which return type the function will return).
5. A function has to be declared in a function declaration at the top of the program and before it can be called by main().

Advantages of Functions:

1. To reduce the size of a program by calling them at different places in the program.
2. Functions can be compiled separately.
3. Using functions, programs are easier to design, debug and maintain.
4. Reuse of the function is possible in multiple programs.
5. The functions provide interfaces to communicate with object.

The function has following elements:

1. A Function Declaration.
2. A Function Definition.
3. A Function Call.
4. Returning from function.

1. A Function Declaration:

- A function declaration tells the compiler about a function's name, return type and parameters.
- This is also referred as function prototyping.
- A function is declared with a prototype, which consists of return type, a function name and argument list.
- **Syntax:**

```
return_type function_name(argument1, argument2, ...);
```

- A function can be declared either in the main function or in a class.
- For example,

```
int add(int a, int b);     //function add with 2 arguments
void func(void);           //function prototype with no arguments
float display();           //function prototype with no arguments
```

2. Function Definition:

- The function definition tells the compiler what task the function will be performed.
- Function definition consists of function prototype and actual body of a function. It can be written anywhere in C++ program.
- The function definition consists of two parts:
  (i) Function Header or Prototype.
  (ii) Function body.
- The function header is similar to declaration.
- The function header is a replica of function declaration. The basic difference between these two is that the declaration in the calling function will end with semicolon whereas; in the called function it is not.
- The function body consists of local variables (if needed) and simple or compound statements.
- **Syntax:**

```
return_type function_name(argument list)
{
    local_variables declaration;
    statements;
}
```

- Here, argument list consists of any number of arguments with any data type or no arguments. The return_type is the type of return value from the function. If we are using return statement, then return_type should be specified otherwise void is used as return_type.
- For example,

```
int add (int a, int b)   //function header
{
    int total;           //local variable
    total = a + b;
    return total;        //function body
}
```

3. A Function Call:

- Function can be called directly by simply writing the function name with arguments or indirectly in the expression if it returns a value.
- We have to pass the actual parameters (arguments) to the functions. In the function definition, the arguments are referenced. Therefore, calling a function is also known as function reference.
- The arguments which we pass should match with the formal parameters which are declared into the function. The data type and the order should also be matched.

- For example,

```
(i)   add (x, y);            //function add called
(ii)  display( );            //function display called
(iii) larger = max(a, b);    //function max called
(iv)  total = add(10, 20);   //function add called
```

4. Returning from Function:

- To return from function we use return statement.
- return statement pushes a value onto the return stack and control is return back to the calling function.
- If return type is void, the function does not return any value.
- **Syntax:**

```
return;  OR
return value;
```

Program 4.1: Program for function.

```
#include<iostream.h>
// using namespace std;
int mult(int a, int b)
{
    int prod;
    prod = a * b;
    return(prod);
}
int main()
{
    int x = 3, y = 4, z;
    z = mult(5, 4);
    cout<<"The first product is="<<z<<"\n";
    cout<<"The second product is="<<mult(x,y)<<"\n";
    cout<<"The third product is="<<mult(5,4);
    return 0;
}
```

Output:

```
The first product is=20
The second product is=12
The third product is=20
Press any key to continue . . .
```

4.3 STATIC MEMBERS

- C++ also allows us to use static members. The members declared inside the class but persisting from their declaration to the end of program are called static members.
- The static members are both data members as well as member functions.
- Static variables defined within a function only have a functions scope so that they may not be accessed by any other function.

4.3.1 Static Data Members

- In C++, a data member of a class can be qualified as static.
- Static variables are normally used to maintain values common to the entire class.
- A static data member is useful when all objects of the same class must share a common item of information.
- Static data members of a class are used to share information among the objects of a class.
- **Syntax for declaring a static data member:**

```
static datatype datamembername;
                OR
datatype static datamembername;
```

- When you proceed a member variables declaration with `static`, you are telling the compiler that one copy of the variable will exist and that of all objects of the class will share that variable.
- **Characteristics of Static Data Member:**
  1. All static variables are automatically initialized to zero value, when the first object of the class is created.
  2. Only one copy of static data member is created for the entire class and is shared by all the objects of that class.
  3. They are also known as class variables.
  4. It is visible only within the class, but its lifetime is the entire program.
  5. Static data members are normally used to maintain values common for all the objects.
  6. The type and scope of each static variable must be redefined outside the class definition. Because static data members are stored separately rather than as a part of an object.

Program 4.2: Program to read name, post and salary of N employees. Post can be manager or supervisor or worker. Add salary of all managers, supervisors, workers and display it.

```
/* Program to illustrate use of static data members in a class */
#include<iostream.h>
#include<conio.h>
#include<string.h>
```

```
class EMP
{
    char name [10], post[10];
    int salary;
    public:
    static int m_sum, s_sum, w_sum; \\ variables for 3 posts
    void get_data( )
    {
        cout<< "\n Enter name, post & salary of an employee =";
        cin>>name;
        cin>>post;
        cin>>salary;
    }
    void add_salary( )
    {
        int x;
        x = strcmp(post, "manager");
        if(x==0)
            m_sum = m_sum + salary;
        x = strcmp(post,"supervisor");
        if(x==0)
            s_sum = s_sum + salary;
        x=strcmp(post,"worker");
        if(x==0)
            w_sum = w_sum + salary;
    }
    void display_data( )
    {
        cout <<"\n"<<name<<post<<salary;
    }
};
// define static variables
int EMP::m_sum;
int EMP::s_sum;
int EMP::w_sum;
```

```
void main( )
{
   EMP E[10];
   int N, i, temp;
   clrscr( );
   cout << "\n Enter total no. of employees = ";
   cin >> N;
   for (i=0;i<N;i++)
   {
      E[i].get_data( );
      E[i].add_salary( );
   }
   cout<<"\n Name Post Salary";
   for (i=0;i<N;i++)
   {
      E[i].display_data( );
   }
   cout<<"\n Total salary of all managers="<<EMP::m_sum;
   cout<<"\n Total salary of all supervisors="<<EMP::s_sum;
   cout<<"\n Total salary of all workers="<<EMP::w_sum;
   temp=EMP::m_sum + EMP::s_sum + EMP::w_sum;
   cout<<"\n Total salary of all the employees in the company = ";
   cout << temp;
   getch( );
   }  // end of main
```

Output:

```
Enter total no. of employees = 4
Enter name, post & salary of an employee =Prajakta manager 10000
Enter name, post & salary of an employee =Pranjal supervisor 8000
Enter name, post & salary of an employee =Vaishali worker 5000
Enter name, post & salary of an employee =Megha worker 40000
Name          Post          Salary
Prajakta      manager       10000
Pranjal       supervisor    8000
Vaishali      worker        5000
Megha         worker        4000
Total salary of all managers=10000
Total salary of all supervisors=8000
Total salary of all workers=9000
Total salary of all the employees in the company = 55000
```

- A static variable can be accessed either using object or using the class name. The scope resolution operator must be used when class name is used to access static data.
- By using static member variables, the need for global variables can be eliminated. Since, static variables are associated with the class itself rather than class object; they are also called as class variables.
- In the above program, observe that the static variables are declared within the class, but defined outside the class. This is necessary to emphasize that the memory space for static data is allocated only once before the program starts execution and is shared by entire class. So this is similar to global data.
- The following diagram makes the concept more clear.

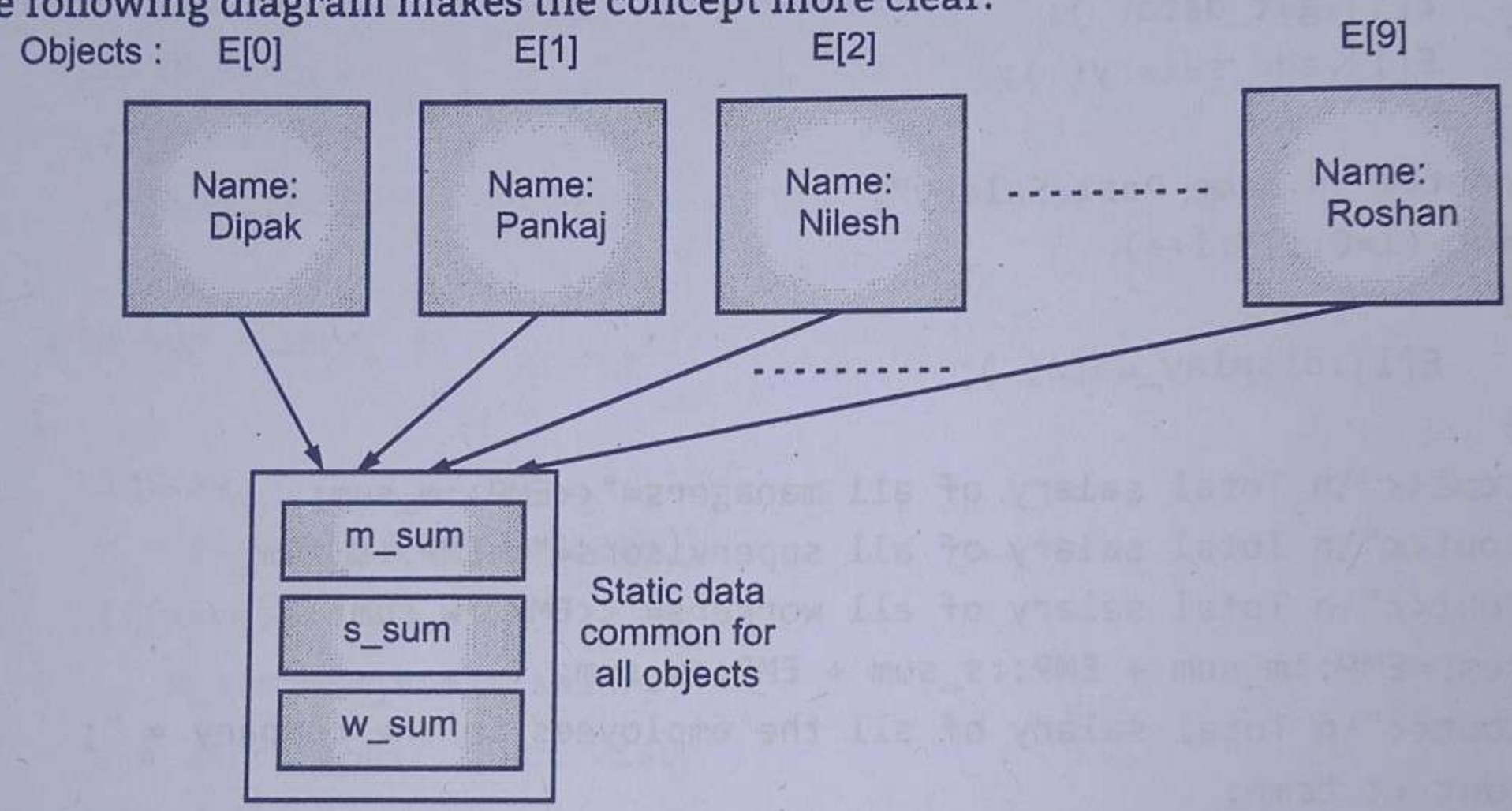


Fig. 4.1: Static Data Members

- Even if the static data members are private, then also it is necessary to define them outside the class. Also note that, the static data members are initialized to zero by default. We can initialize them to any value at the time of definition.

  For example: `int EMP:: m_sum = 100;`
- This statement will initialize m_sum to 100 instead of zero.
- One more use of static data members is to keep count of total number of objects created for that class.

4.3.2 Static Member Functions

- Member functions may also be declared as static.
- A static function can have access to only other static members declared in the same class.
- But there are some restrictions on them.
  1. They may only access other static members of the same class directly.
  2. They do not have a 'this' pointer.
  3. There cannot be a static and a non-static version of the same function.

- It can be called using class name as,

```
class_name:: function_name
```

- **Syntax of declaring a static member function:**

```
static return_type function_name (arguments);
                OR
return_type static function_name(arguments);
```

- **Syntax for calling static member function:**

```
class_name::member-function-name;
```

Program 4.3: Display a class which displays the number of times display operation is performed.

```
#include<iostream.h>
class display
{
   private:
      static int count;
   public:
      static void disp()
      {
         count++; //increment cout
         cout<<count<<endl;
      }
};
int display:: count; //defining static variable outside a class
int main()
{
   display ob1,ob2,ob3,ob4;
   ob1.disp();
   ob2.disp();
   ob3.disp();
   ob4.disp();
   //function call with class name
   display:: disp();
}
```

Output:

```
1
2
3
4
5
Press any key to continue . . .
```

- The statement count++ is executed when disp() function is invoked and the current value of count is assigned to each object.

4.4 INTERACTION BETWEEN FUNCTIONS

- The function [main() and other] interacts with each other through parameters. We pass the parameter to the function for the communication of calling function and the called function.
- There are two types of parameters actual parameters (used in the function call) and formal parameters (used in function definition).

4.4.1 Call by Reference

- When a function is called the address of the actual parameters are copied on to the formal parameters, though they may be referred by different variable names.
- This content of the variables that are altered within the function block are return to the calling function program. As the formal and the actual arguments are referencing the same memory location or address.
- This is known as call by reference, when we use this concept with functions.
  - Called function makes the alias of the actual variable which is passed.
  - Actual variables reflect the changes made on the alias of actual variables.
  - Number of values can be modified together.
  - For call by reference concept C++ uses a reference operator (&).
  - '&' operator allows true call by reference functions, eliminating the need to dereference arguments passed to the function.
  - The '&' operator is placed before the variable name of argument in the parameter list of the function. Then the function can be called without passing the address and without dereferencing within the function.

Program 4.4: Program to swap the values by call by reference.

```
#include<iostream.h>
void swapr (int &p, int &q)
{
    int dummy;
    dummy = p;
    p = q;
    q = dummy;
}
int main()
{
    int x = 225, y = 305;
    cout<<x<<"\t"<<y<<endl;
    swapr(x,y);
    cout<<"After swap"<<endl;
    cout<<x<<"\t"<<y;
}
```

Output:

```
225      305
After swap
305      225
```

- Let us see how this program gets executed in the memory. This is shown by Fig. 4.2.
- In the Fig. 4.2, p is alias of x and q is alias of y. Therefore, we do not need 4 bytes allocated for p and q, [each will have 2 bytes].
- Here, it first copies the value of x i.e. p to dummy. Then the value of q or y is assigned to p and then the value of dummy is assigned to y or q. In this way the values are swapped without wasting more amount of memory. This is because the formal parameters are of reference type.
- The compiler always treats them similar to pointers but compiler does not allow the modification of the pointer value. The changes made to the formal parameters p and q are reflected on the actual parameters x and y as shown in Fig. 4.2.

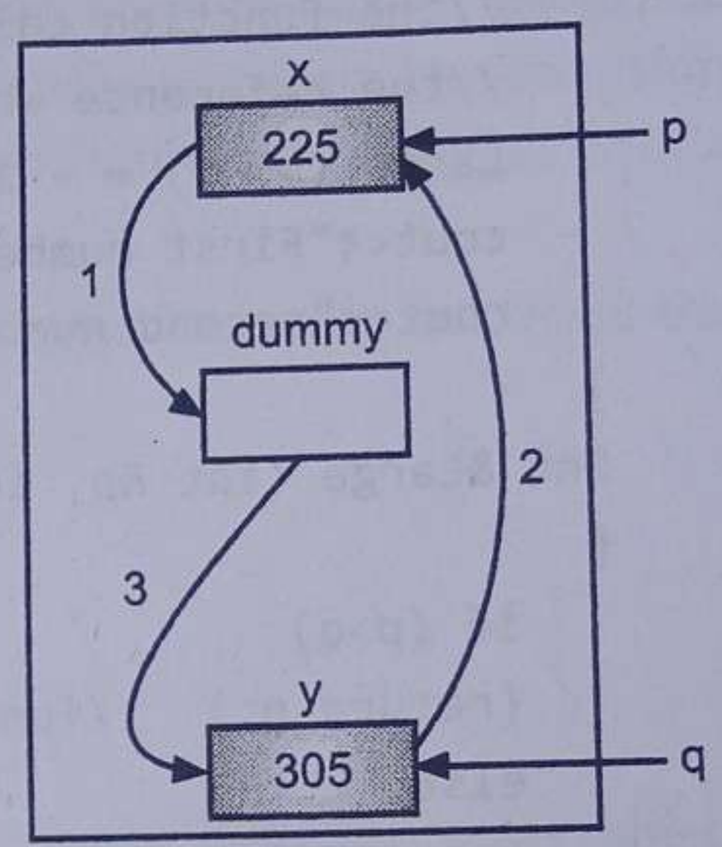


Fig. 4.2: Call by Reference

- The following points can be noted about reference parameters:
  1. Here, parameters passed to a function by its reference and not by its value. Reference means address of variable is passed to function.
  2. A reference cannot be null.
  3. It should always refer to an object or a variable.
  4. Once, a reference points towards one object or variable, then it cannot be changed. Even it should not point towards different object.
  5. The reference does not require any explicit mechanism to dereference the memory. Whenever the program execution terminates, all the memory locations gets free. References are also lost.
  6. It is good for performance because it eliminates the overhead of copy large amount of data.

4.4.2 Return by Reference

- The function can also return the reference of variable. This reference variable is actually an alias for the referred variable.
- The method of returning reference is used generally in operator overloading. This method will form a cascade of member function calls.

  For example: `cout << k << a;`
- This statement is a set of cascaded calls which returns reference to the object `cout`.

Program 4.5: Program for function which returns value by reference.

```
#include<iostream.h>
int &Large (int &m, int &n);    //function prototype
int main()
{
    int a,b,m,n;
    cout<<"Enter values for two integers";
    cin>>a>>b;
    //the function call is given
    //the reference which is returned is assigned to - 1
    Large (m, n) = - 1;
    cout<<"First number"<<m;
    cout<<"second number"<<n;
}
int &Large (int &p, int &q)
{
    if (p>q)
    {return p;}  //function returns the value through reference
    else
    {return q;}
}
```

Output:

```
Enter values for two integers    10    20
First number     10
Second number    - 1
If we executive the program one more time then output will be:
Enter values for two integers    8    3
First number     - 1
Second number    3
```

- When the program gets executed, it returns the reference to the variable which holds the larger value. And this assigns the value - 1 to it.

4.5 INLINE FUNCTIONS [S-19]

- C++ inline function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.
- Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

- The advantage of using function is to save the memory space. Consider a situation where a small function is called number of times calling a function requires certain overheads such as:
  - Passing to the function.
  - Passing the values or push arguments into the stack.
  - Save the registers.
  - Return back to calling function.
- In C++, compiler put the code directly inside the function body of calling program. Every time when function is called, at each place the actual code from the function would be inserted, instead of a jump to the function. Such functions are called inline functions.
- A function can be defined as an inline functions by writing the keyword `inline` to the function header as shown below:

```
inline function_name (list of arguments)
{
    //function body
}
```

- When we do not specify keyword `inline` then only one copy of function code (object code) is called depending upon the functions calls. This is illustrated in Fig. 4.3.

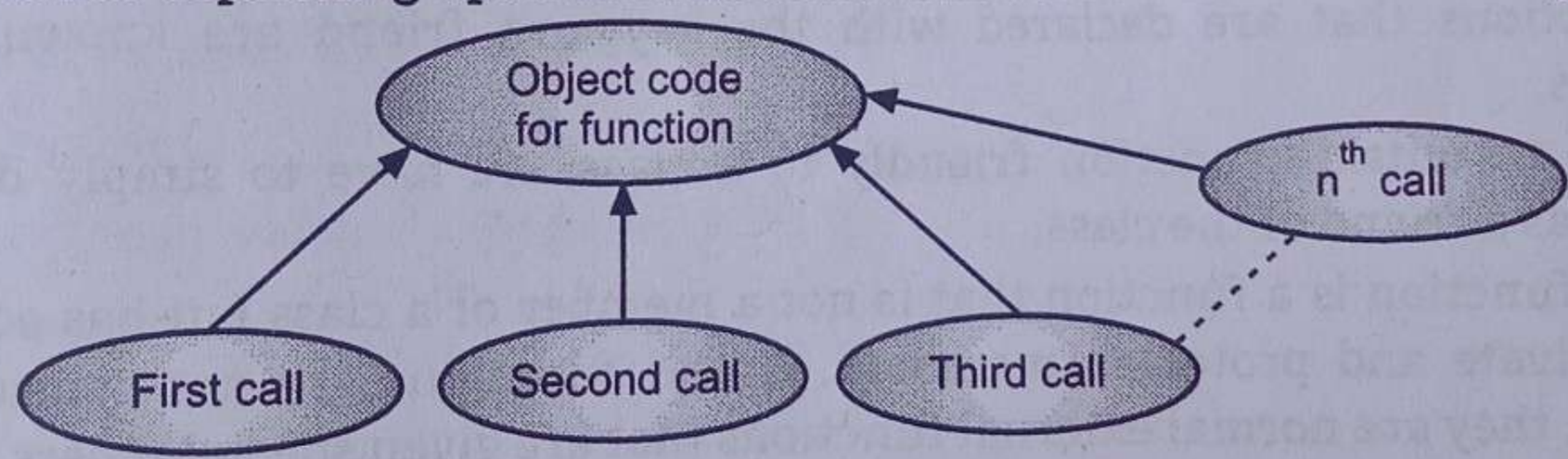


Fig. 4.3: Calls Given to a Function

- But as we said above, lot of time is needed for all these calls to execute. If we make this function inline then the calling statement is replaced with the function code. So it minimizes the time spent on jumping, pushing arguments and so on. This is shown in Fig. 4.4.

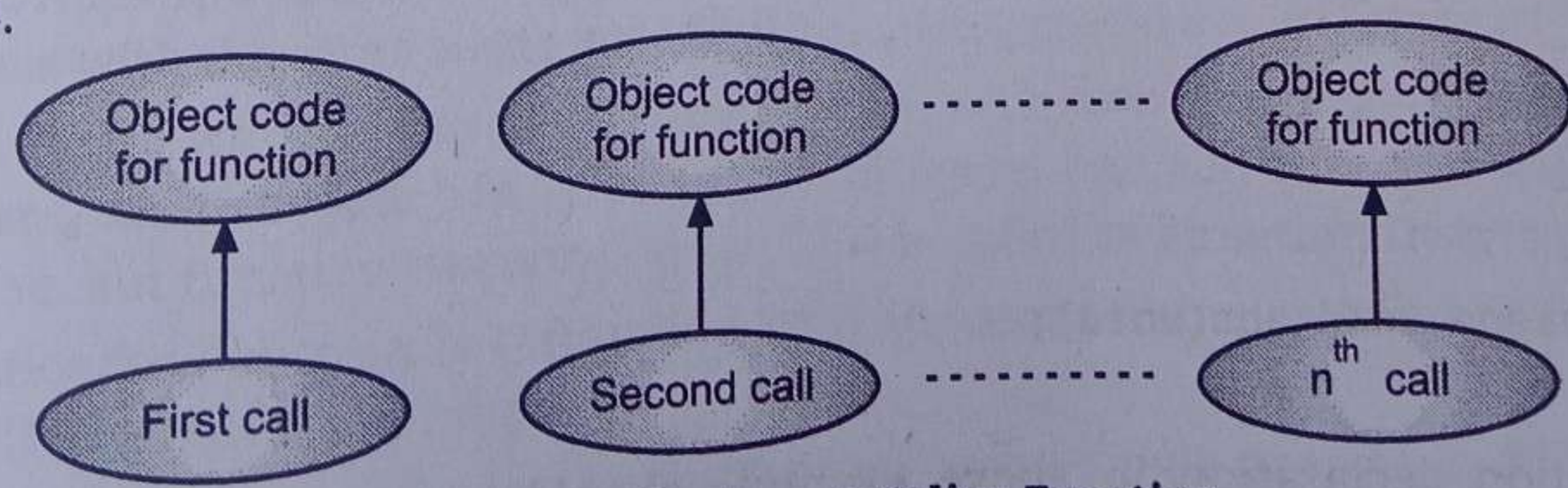


Fig. 4.4: Calls given to Inline Function

Program 4.6: Program illustrates the use of inline function to compute square of a number.

```
#include<iostream.h>
inline int sq (int x)
{
    int p;
    p = x * x;
    return p;
}
int main()
{
    int n;
    cout<<"Enter number";
    cin>>n;
    cout<<"square is"<<sq(n)<<endl;
}
```

Output:

```
Enter number 12
square is 144
```

4.6 FRIEND FUNCTION [S-18, 19, W-18]

- The functions that are declared with the keyword friend are known as Friend Functions.
- To make an outside function friendly to a class, we have to simply declare this function as a friend of the class.
- A friend function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges.
- Friends are not in the class's scope, and they are not called using the member selection operators (. and –>) unless they are members of another class.
- **For example:**

```
class xyz
{
    ......
    public:
    ......
    ......
    friend void abc(void);
};
```

- The function declaration in above example should be preceded by the keyword friend.

Program 4.7: Program for friend function.

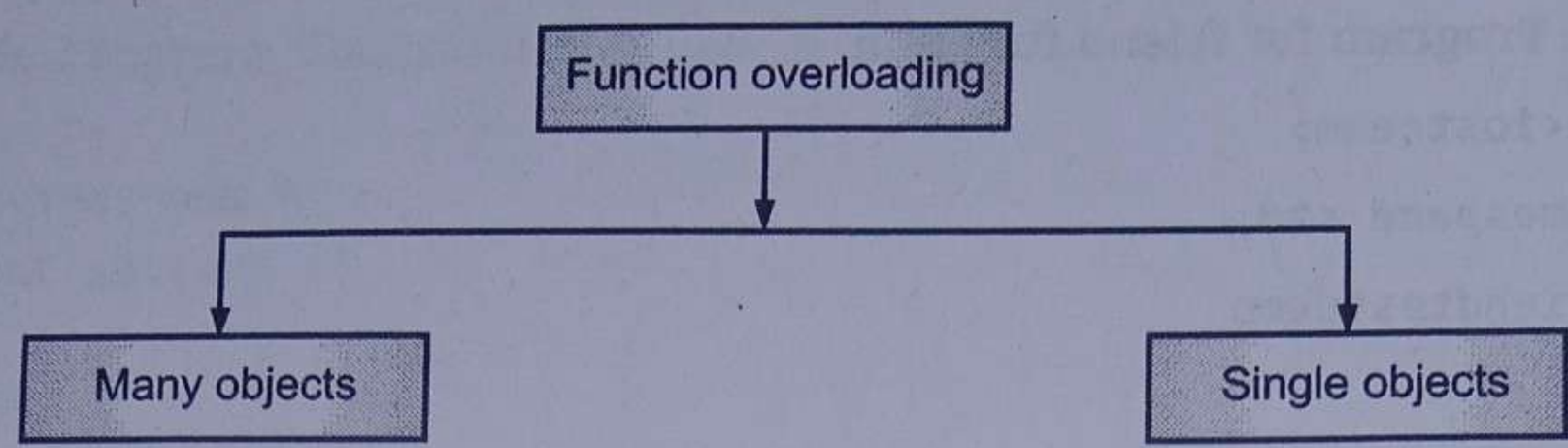
```
#include<iostream>
using namespace std;
class friendtestdemo
{
   int x;
   int y;
 public:
   void setvalue() {x = 20; y = 50;}
   friend float mean(friendtestdemo f);
};
float mean(friendtestdemo f)
{
   return float(f.x + f.y)/2.0;
}
int main()
{
   friendtestdemo A;
   A.setvalue();
   cout<<"Mean value="<<mean(A)<<"\n";
   return 0;
}
```

Output:

```
Mean value = 35
```

4.7 FUNCTION OVERLOADING [W-18, S-18]

- C++ provides the facility of function overloading in which we can define multiple functions with the same name but the types of argument are different.
- For example, we can have a function add with two floating point numbers as argument, another function adds with 3 integers, one add function with 4 doubles and so on. But function name is similar. This is called as **Function Overloading**.
- An overloaded function is a function with the same name as another function, but with different parameter types.
- The function overloading can be performed using single and many objects. This is shown in following Fig. 4.5.



- Same function name with different argument list.
- Different objects are created.
- A call is given to different arg-list functions with each object.

- Same function name with different argument list.
- One object is created.
- A call is given to different arg-list function with one object.

Fig. 4.5: Function Overloading

- **Rules for overloaded functions:**
  1. The argument list of each of the function instances must be different.
  2. The compiler does not use the return type of the function to distinguish between function instances.
- The Program 4.8 explains the concept of function overloading with many objects and Program 4.9 explains the concept with one (single) object.

Program 4.8: Program for function overloading with many objects.

```
#include<iostream.h>
class test
{
   public:
   void addnum (int p, int q, int r)
    {
      cout<<"sum is"<<p+q+r<<"\n";
    }
   void addnum (float p, float q)
   {
      cout<<"sum is"<<p + q <<"\n";
    }
   void addnum(int p, double q)
   {
      cout<<"sum is"<<p + q<<"\n";
   }
};
```

```
int main()
{
    test ob1, ob2, ob3;
    ob1.addnum (5, 8, 10);
    ob2.addnum (2.5, 3.6);
    ob3.addnum (25, 3.5);
}
```

Output:

```
sum is 23
sum is 5.6
sum is 28.5
Press any key to continue . . .
```

Program 4.9: Program for function overloading with single object.

```
#include<iostream.h>
class test
{
    public:
    void addnum(int p, int q, int r)
    {
     cout<<"sum is"<<p+q+r<<"\n";
    }
    void addnum (float p, float q)
    {
     cout<<"sum is"<<p+q<<"\n";
    }
    void addnum(int p, double q)
    {
    cout<<"sum is"<<p+q<<"\n";
    }
};
int main()
{
    test ob1;
    ob1.addnum (5, 8, 10);
    ob1.addnum (2.5, 3.6);
    ob1.addnum (25, 3.5);
}
```

Output:

```
sum is 23
sum is 5.6
sum is 28.5
Press any key to continue . . .
```

4.8 CONSTRUCTORS [S-19, W-18]

- A constructor is a special member function which is used to allocate memory space and values to the data members of that object.
- A constructor makes the object functional by converting an object with the unused (uninitialized) memory into a usable (initialized) object.
- Whenever, an object is created, the constructor will be executed automatically and initialization of data member takes place.
- It is called constructor because it constructs the value of data member of the class.
- A constructor is a special as it has the same name as that of the class and is automatically invoked whenever an object of the class is created.
- When we declare an object in function main( ):
  1. Memory will be allocated.
  2. Constructors will be called by the compiler automatically.
  3. Initialization of variables (data member) takes place.
- It is not possible for us to initialize the data member in the class declaration itself. Because at that point memory is not allocated for the object.

4.8.1 Syntax Rules for Writing Constructor Functions

- The following are the rules used for writing a constructor. These are also referred as characteristics of constructors.
  1. A constructor's name must be same as the class name in which it is declared.
  2. It does not have any return value.
  3. It is generally declared as a public member function. It may have protected access within the class and only in rare circumstances it should be declared private.
  4. It can have default argument.
  5. It can not be declared constant, variable, static or virtual.
  6. It can not be referred by its address.
  7. It can not be inherited like other member functions.
  8. The implicit calls are given to the operators new and delete when memory allocation is required.
  9. It is automatically called when an object is created.
  10. It specifies how an object is created.
- Like other member functions of the class, a constructor can also defined either inside or outside the class definition.

- **Syntax to define a constructor (Inside the class):**

```
class class_name
{
    .
    .
    public:
        class_name(parameter_list)        //header of the constructor
        {
            //body of the constructor ......
        }
};
```

Where, paramter_list is optional.

- **Example:**

```
//class with constructor
class time
{
    int hour;
    int min;
    public:
        time()
        {
            hour = min = sec = 0;  //initialization
        }
}
```

- A constructor can also be declared outside the class using the scope resolution operator (: :).
- **Syntax to define the constructor (Outside the class):**

```
class class_name
{
    .
    .
    public:
        class_name(parameter_list); //constructor prototype
        .
        .
};
class_name::class_name(parameter_list)//constructor definition
{
    //body of the constructor ......
}
```

Where, paramter_list is optional.

- **Example:**

```
//class with constructor
class time
{
    int hour;
    int min;
    public:
        time();                //constructor declared
}
time::time()                   //definition of constructor
{
    hour = min = sec = 0; //initialization
}
```

- When a class is declared with the constructor as above, then object is created by the class will be initialized automatically. For example: `time t;`
- Here, object `t` is created and it initializes data members: `hour, min, sec` to zero. There is no need to write any statement to invoke the constructor function (as we do with normal function).

Program 4.10: Program for constructor.

```
#include <iostream>
using namespace std;
class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();  // This is the constructor
   private:
      double length;
};
Line::Line(void)
{
   cout << "Object is being created" << endl;
}
void Line::setLength( double len )
{
   length = len;
}
```

```
double Line::getLength( void )
{
   return length;
}
int main()
{
   Line line;
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;
   return 0;
}
```

Output:

```
Object is being created
Length of line : 6
```

4.8.2 Constructor Call

- For calling of a defined constructor there are two ways namely, explicit call, and implicit call.
- If the name of the constructor is not used in the object declaration, the call is known as an implicit call to the constructor. While, if the name of the constructor is used in the object declaration, the call is known as an explicit call to the constructor.

1. **Explicit Call:** In this method name of constructor with the values of variables in the bracket are coming in the picture, to construct the values. Name of constructor is explicitly mentioned in the program, so this method is known as explicit call of constructor.

   For example:

   ```
   Bank b1;
   b1 = Bank(12);
   ```

2. **Implicit Call:** It will not show name of constructor in the program. Values of the variables get constructed at the time of objects declaration.

   For example: `Bank b1(50);`

Program 4.11: Program for calling of constructor implicitly and explicitly.

```
#include<iostream>
using namespace std;
class display
{
   int a, b;
```

```
    public:
        display()  //constructor
        {
            a=1;
            b=2;
            cout<<"Value of a and b is : "<<a<<", "<<b<<endl;
        }
};
int main()
{
    display disp;        //implicit call
    display displ = display();//explicit call
    return 0;
}
```

Output:

```
Value of a and b is : 1, 2
Value of a and b is : 1, 2
```

4.8.3 Types of Constructor

- Fig. 4.6 shows types of constructors in C++.

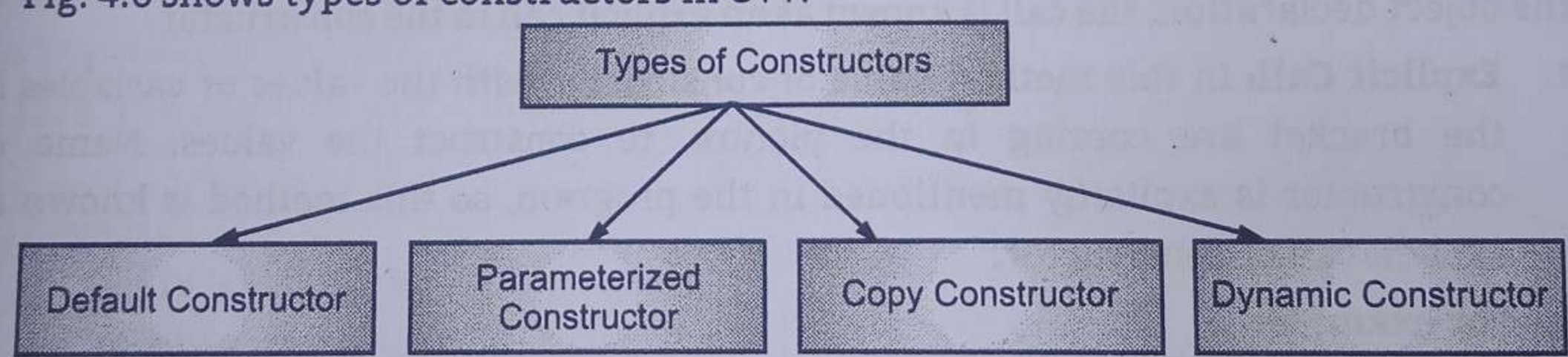


Fig. 4.6: Various Types of Constructors

4.8.3.1 Default Constructor

- The constructor which does not accept any argument is called default constructor. In default constructor the argument list is void.
- Default constructor is also called as empty constructor because of it has no arguments. A default constructor is used to initialize all the objects of a class with the same values.
- There can be only one default constructor in a class. The default constructor is defined as, "a constructor which does not contain any parameters/arguments".
- A default constructor can be called directly when an object of the class is created. If no constructor is created, compiler will create a default constructor by itself.

- **Syntax:**

```
class class_name();
{
    .
    .
    .
    public:
        class_name()
        {
            //body of constructor
        }
    .
    .
    .
};
```

- **For example:**

```
student:: student()
{
    rollno=0;
    marks=0.0;
}
```

- The diagrammatic representation of default constructor is shown in Fig. 4.7.

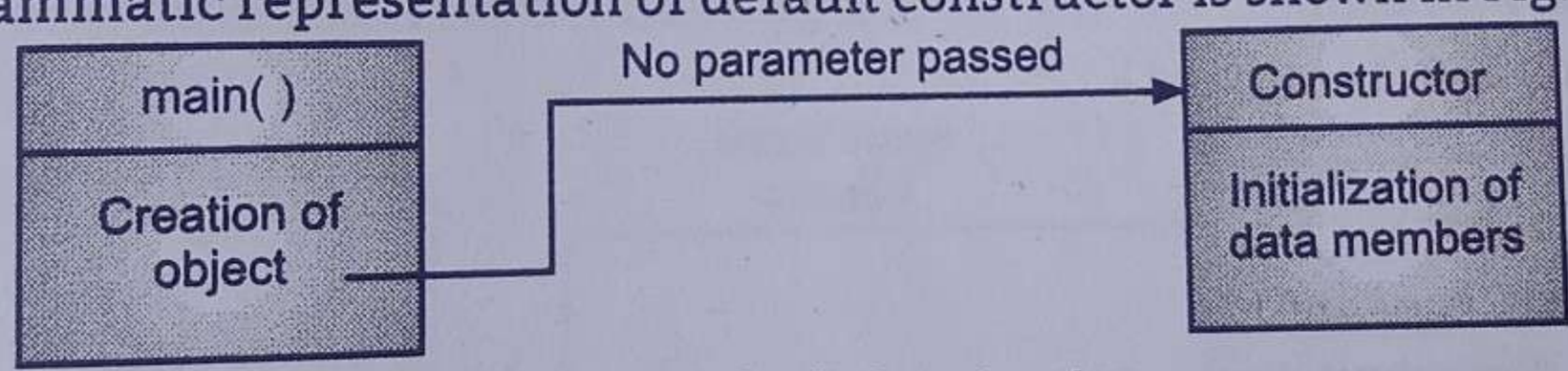


Fig. 4.7: Default Constructor

Program 4.12: Program for generation of fibonacci series using default constructor and scope resolution operator.

```
#include<iostream>
using namespace std;
class fibo
{
    private:
        int fib0, fib1, fib2;
    public:
        fibo(); // default constructor declaration
        void disp();
        void increment();
};
```

```
// outside class scope resolution operator is used
fibo:: fibo()
{
    fib0 = 0;
    fib1 = 1;
    fib2 = fib0 + fib1;
}
void fibo:: increment()
{
    fib0 = fib1;
    fib1 = fib2;
    fib2 = fib0 + fib1;
}
void fibo:: disp()
{
    cout<<fib2<<"\t";
}
int main()
{
    fibo ob1;
    for (int i=0;i<=15;++i)
    {
        ob1.disp();
        ob1.increment();
    }
    return 0;
}
```

Output:

```
1   2   3   5   8   13   21   34   55   89   144   233   377   610   987   1597
```

4.8.3.2 Parameterized Constructor

- Sometimes, it is essential to initialize the various data elements of different objects with different values when they are created. This is achieved by passing arguments to the constructor function when the objects are created.
- The constructors which accept any number of formal parameters and that formal parameters are used to initialize the object are called parameterized constructors.
- A parameterized constructor is defined as, "a constructor which accepts one or more arguments/parameters at the time of declaration of objects and initializes the data members of the objects with these parameters/arguments".

- **Syntax to define a parametrized constructor:**

```
class class_name();
{
    .
    .
    .
  public:
      class_name() (param1, param2, ... paramN)
      {
          //body of the constructor ...
      }
      .
      .
      .
};
```

- **For example:**

```
student:: student(int r)
{
    rollno=r;
}
```

- These parameters are passed at the time of object creation. These parameters can be of any data type except its own class data type. The diagrammatic representation is shown in Fig. 4.8.

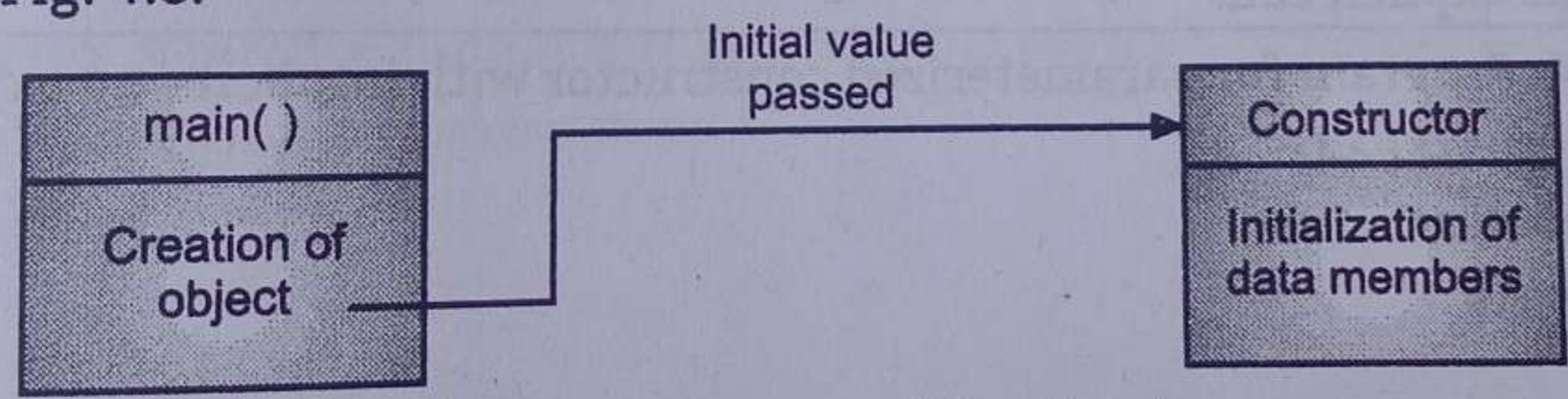


Fig. 4.8: Parameterized Constructor

Program 4.13: Program for parameterized constructor. [S-18]

```
#include<iostream>
using namespace std;
class sample
{  private:
      int k;
   public:
      sample (int p)        // parameterized constructor
      {
          k = p;
      }
```

```
        void disp ()
        {
            cout<<k;
        }
};
int main ()
{
    sample ob1(15); // value passed through object
    ob1.disp();
    return 0;
}
```

Output:

```
15
```

- The output of above program is 15. The formal parameter value is passed to the constructor where actual parameter gets this formal value and initialization of object is done.
- The passing of argument can be done in following two ways:
  1. Call a constructor explicitly.
  2. Call a constructor implicitly.
- Program 4.14, is the example of implicit call. Here the values are passed through the objects. Sometimes, the constructor itself consists of certain default values. If the values are passed to the constructor, it will consider the default values. This is referred as explicit call.

Program 4.14: Program for parameterized constructor with implicit and explicit call.

```
#include<iostream>
using namespace std;
class sample
{
    private:
        int k;
    public:
        sample (int p)
        {
            k = p;
        }
        void disp ()
        {
            cout<<k;
        }
};
```

```
int main()
{
    sample ob1 (10); // implicit
    sample ob2 = sample (20);  // explicit
    ob1.disp();
    ob2.disp();
    return 0;
}
```

Output:

```
10 20
```

- In above program, ob1 shows implicit call whereas ob2 shows explicit call. Implicitly the default value of constructor is assigned to ob2.

4.8.3.3 Copy Constructor [S-19, W-18]

- A copy constructor takes a reference to an object of the same class as itself as an argument.
- A constructor that initializes an object using values of another object passed to it as parameter is called copy constructor.
- In C++, a new object of a class can also be initialized with an existing object of the same class. For this, the compiler of C++ calls the copy constructor. Copy constructor creates the copy of the passed object.
- A copy constructor is defined as, "a constructor which accepts an already existing object through reference to copy the data member values".
- **Syntax to define a copy constructor:**

```
class class_name
{
    .
    .
    .
    public:
        class_name(class_name & object_name)
        {
            // body of the constructor ........
        }
    .
    .
    .
};
```

- As with all constructors, the function name must be the class name. The parameter in the declaration is a reference to the class, which is a characteristic of all copy constructors.
- While defining the copy constructor, it is recommended to specify the keyword const for the reference parameter. This is because the existing object is used only to initialize the new object and cannot be changed by the copy constructor.
- **Syntax:**

```
class class_name
{
    .
    .
    .
  public:
    class_name(const class_name & object_name)
    {
        // body of the constructor
    }
    .
    .
    .
};
```

- **To invoke a copy constructor use following syntax:**

```
class_name new_object = existing_object;   OR
class_name new_object (existing_object);
```

- The existing object is the object whose copy is to be created and stored in new object. The argument is passed through reference and not through value.

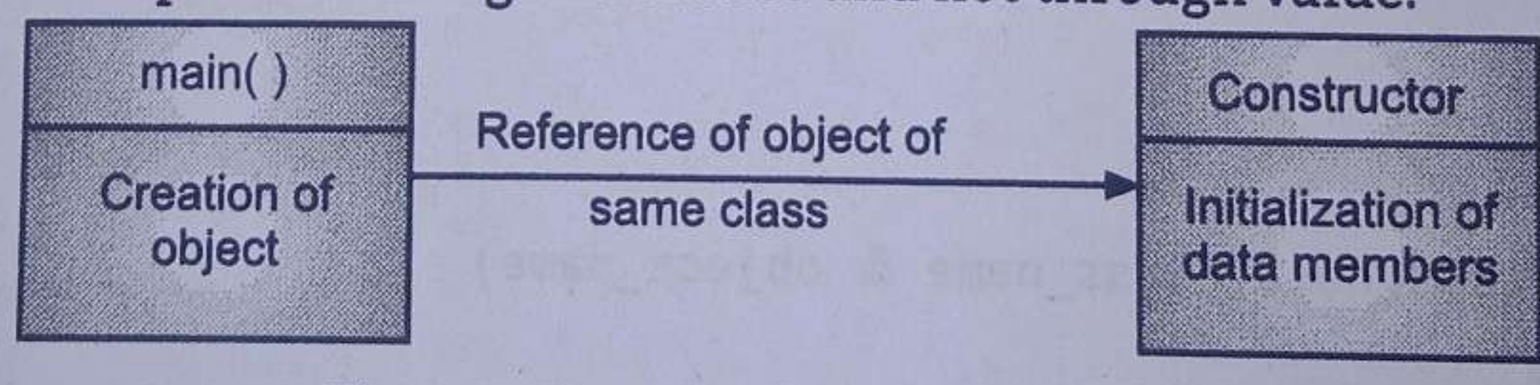


Fig. 4.9: Concept of Copy Constructor

- **For example:**

```
student:: student(student &t)
{
    rollno = t.rollno;
}
```

Program 4.15: Program for copy constructor.

```
#include<iostream>
using namespace std;
class sample
{
   private:
         int x, y;
   public:
         sample();                    //default constructor
         sample(sample &);            //copy constructor definition
         void disp();
};
sample::sample()
{
   x = 10;
   y = 20;
}
sample:: sample (sample &s)
{
   x = s.x;          //existing object and data
   y = s.y;
}
void sample:: disp()
{  cout<<x<<" "<<y<<endl; }
int main()
{
   sample ob1;
   sample ob2(ob1);     //invoking copy constructor
   sample ob3 = ob1;       //invoking copy constructor
   ob1.disp();
   ob2.disp();
   ob3.disp();
}
```

Output:

```
10 20
10 20
10 20
```

4.8.3.4 DYNAMIC CONSTRUCTOR [S-18]

- Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.
- The memory is allocated with the help of the new operator. The constructor can also be used to allocate memory while creating objects.
- This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in saving of the memory.
- The word dynamic is related with the memory allocation. Programmer can explicitly assign memory to variables and after completion of operation we can delete that memory.
- 'new' is the keyword used for assigning memory locations to variables while 'delete' is the keyword used for deleting memory locations. By using these two keywords memory can be effectively managed. The constructor can also be used to allocate memory while creating objects. For this purpose new and delete keywords are allowed in the body of constructor.
- A dynamic constructor is defined as, "a constructor in which the memory for data members is allocated dynamically". Dynamic constructors are used to allocate memory to objects at run-time rather i.e., allocating memory to objects when they are created.

Program 4.16: Program for Dynamic Constructor.

```
#include <iostream>
using namespace std;
class Dynamic
{
    int *ptr;
    public:
      Dynamic()
      {
          ptr=new int;
          *ptr=1000;
      }
      Dynamic(int v)
      {
          ptr=new int;
          *ptr=v;
      }
```

```
        int Display()
        {
            return(*ptr);
        }
};
int main()
{
    Dynamic d1;
    Dynamic d2(2000);
    cout<<d1.Display();
    cout<<d2.Display();
}
```

Output:

```
1000
2000
```

Program 4.17: A program to create memory space for a class object using the new keyword and to destroy it using delete keyword.

```
#include<iostream>
using namespace std;
class sample
{
    private:
            int x;
            float y;
    public:
            void getdata();
            void display();
};
void sample:: getdata()
{
    cout<<"Enter an integer value \n";
    cin>>x;
    cout<<"Enter float value \n";
    cin>>y;
}
```

```
void sample:: display()
{
    cout<<"x="<<x<<"\t"<<"y="<<y;
}
int main()
{
    sample *ptr;            // pointer declaration
    ptr = new sample;       // dynamic memory allocation
    ptr -> getdata();
    ptr -> display();
    delete ptr;             // releasing allocated memory
}
```

Output

```
Enter an integer value
7
Enter float value
7.7
x = 7  y = 7.7
```

4.9 DESTRUCTOR

- Destructor is a special member function which is called automatically whenever a class object is destroyed.
- The primary usage of the destructor function is to release space on the heap. Whenever, a particular object goes out of the scope of its existence, destructors will be called automatically and it takes out the allocated memory.
- Destructors are invoked by the compiler implicitly when the termination of program takes place. But it is always better to invoke the destructor explicitly by writing it into the program code.
- The diagrammatic representation of the destructor is shown in Fig. 4.10.

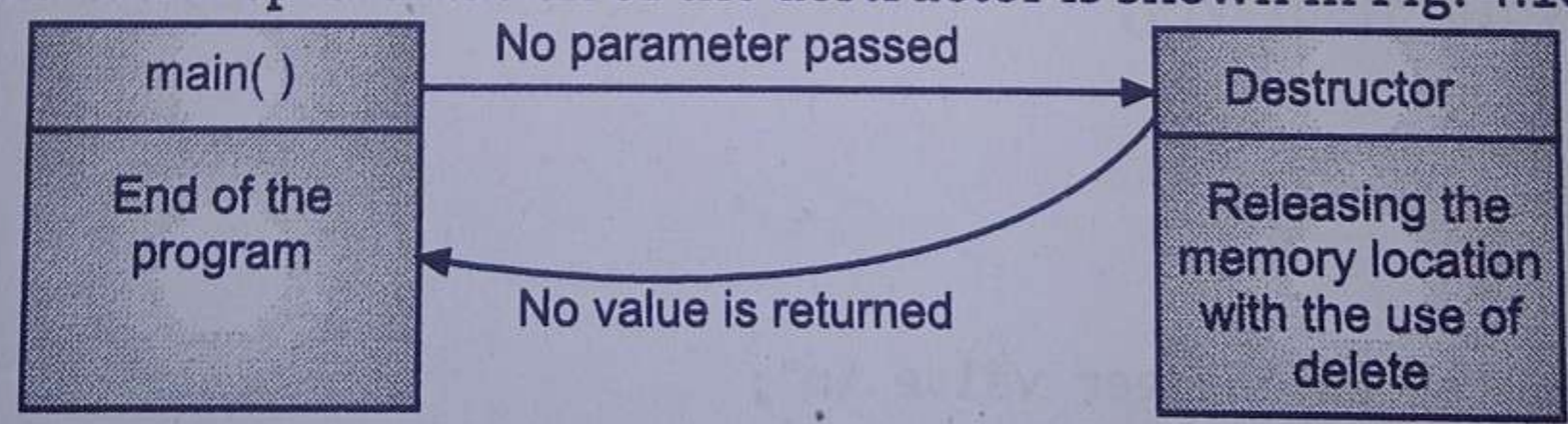


Fig. 4.10: Destructor

- A destructor is also special as it has same name as that of the class of which it is a member but with a ~ (tilde) sign prefixed to its name and which is used to destroy the objects that have been created by a constructor. It gets invoked when an object's scope is over.

- **Syntax of Destructor:**

```
class class_name
{
    .
    .
    .
    public:
        class_name(); { }
        ~class_name()    //header of the destructor
        {
            //body of the destructor.....
        }
        .
        .
        .
};
```

- **For example,**

```
class sample
{
    private:
        //data variables
        //member functions
    public:
        sample()      //constructor
        ~sample() //destructor
        //member functions
};
```

- A destructor releases the resources and memory at run-time to clean up the unused storage area.
- **Syntax Rules for Writing Destructor Functions:**
  1. A destructor name must be same as the class name in which it is declared, prefixed or preceded by a symbol tilde (~).
  2. It does not have any return value.
  3. It is declared as a public member function.
  4. It takes no argument and therefore cannot be overloaded.
  5. It cannot be declared static, constant, variable or volatile i.e. virtual.
  6. It is automatically called when the object is destroyed.
  7. It specifies how an object is deleted.

Program 4.18: Program for destructor.

```
#include<iostream>
using namespace std;
class sample
{
    private:
        int k, m;
    public:
        sample()
        {
            k = 10;
            m = 20;
            cout<<k<<"\t"<<m<<endl;
        }
        ~ sample() {    }   // destructor
};
void main()
{
    sample ob1, ob2, ob3;
}
```

Output:

```
10      20
10      20
10      20
```

- The objects deleted are:
  ob3
  ob2
  ob1
- If more than one object is defined, constructors are invoked from **right to left** i.e., from first object to last. But destructors are invoked in reverse order like a stack. So first created object will get deleted last.

Program 4.19: Program to declare class having data members as hrs. mins. and secs. Write constructor the values and destructor to destroy values. Accept and display data for two objects.

```
#include<iostream>
using namespace std;
class time
{
    int hr;
    int min;
    int sec;
```

```
public:
    time (int h, int m, int s)
    {
        hr = h;
        min = m;
        sec = s;
    }
    void display()
    {
        cout<<"Time = "<<endl;
        cout<<hr<<": "<<min<<":"<<sec;
    }
    ~time ()
    {
        cout<<"Destructor invoked";
    }
};
int main()
    {
        time t1(3, 30, 50);
        time t2(4, 15, 35);
        t1.display();
        t2.display();
        return 0;
    }
```

Output:

```
Time = 3: 30: 50
Time = 4: 15: 35
Destructor invoked
Destructor invoked
```

- In above program t2 object will get destroyed first and then t1 object will destroyed because destructors are called in opposite direction of constructors.

Difference between Constructor and Destructor:

Table 4.1: Difference between Constructor and Destructor

Terms	Constructor	Destructor
Purpose	It allocates the memory to an object.	It deallocates the memory of an object.

Declaration Syntax	class_name (arguments if any)
	{
	//Body of Constructor...
	};
		~class_name ()
		{
		//Body of Destructor...
		};

Arguments	Constructor accepts arguments.	Destructor does not accept any argument.
Calling	Constructor is called automatically, when a new object of a class is created.	Destructors are invoked/called by the compiler implicitly when the termination of program takes place.
Name	Constructor has the same name as class name.	Destructor also has the same name as class name but with a tilde (~) prefixed to its name.
Use	Constructor is used for initializing the values to the data members of the class.	Destructor is used when the object is destroyed or goes out of scope.
In numbers	There can be multiple constructors in the class.	There is always a single destructor in the class.
Overloading	Constructors can be overloaded.	Destructor cannot be overloaded.
Return type	Constructors never have a return type even void.	A destructor neither accepts any parameter nor has a return type (not even void).
Types	In C++, program we can use different types of constructors like default constructor, parameterized constructor, copy constructor etc.	In C++, program a single destructor is used to destroy all the objects of a class created either using default, parameterized or copy constructor.

Example:	class Student
	{
	private:
	int marks;
	char grade;
	public:
	Student(int m, char g)
	{
	marks= m;
	grade= g;
	}
	void show()
	{
	cout<<"Marks ="<<marks<<endl;
	cout<<"Grade = "<<grade<<endl;
	}
	};
		class test
		{
		private:
		int a;
		public:
		test()
		{
		cout<<"object is created
		:"<<endl<<endl;
		}
		~test()
		{
		cout<<"Object is
		destroyed"<<endl<<endl;
		}
		};

4.10 OPERATOR OVERLOADING [W-18, S-18]

- Operator overloading is closely related to function overloading.
- Operators can be overloaded by creating **'operator functions'**.
- An operator function defines the operations that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword **'operator'**.
- A operator function's general form is:

```
return_type class_name:: operator op (arg-list)
{
    // operations
}
```

Where, return-type is the type of value returned by the specified operation and op is the operator being overloaded.

- For example, if you are overloading the / operator, use 'operator /'. When a unary operator is overloaded, arg-list is empty.
- When a binary operator is overloaded, arg-list will contain one parameter.
- Operators that can be overloaded except the following:

.	Member access operator
.*	Pointer to member operator
::	Scope resolution operator
?:	Conditional operator
sizeof	sizeofoperator.

- All operators are divided into three categories:

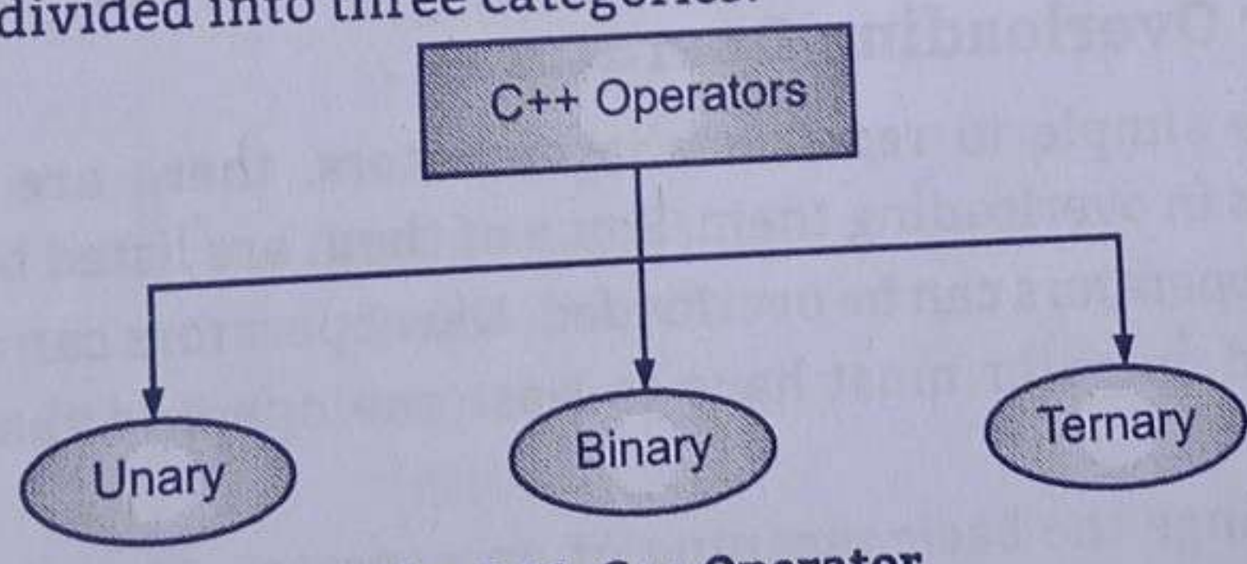


Fig. 4.11: C++ Operator

1. **Unary:**
   It requires only one operand to perform operation. For example: i++
   Here, i is the single operand to perform increment operation.
2. **Binary:**
   It requires two operands to perform operation. For example: a + b;
   Here, a and b are two operands to perform addition operation.

3. Ternary operator:

It requires three operands to perform operation. Ternary operator is?,:

For example: `x > 5? y = 2: y = 1;`

This line compares x value. If x > 5, y = 2. If x is not greater than y = 1.

- Overloading means assigning more than one meaning to operator.
- For example, + sign can be used for:
  1. Addition of two integers.
  2. Addition of two complex numbers.
  3. Addition of two characters.
- **Steps for operator overloading are given below:**
  1. Create a member function under a public section.
  2. The member function is called as operator overloaded function so, syntax of declaration than other normal member function.

     Syntax:

     ```
     return_type operator op( )
     {
        //details of operation
     }
     ```

     Here, operator is a keyword which tells it is operator overloading function and op is any operator. For example, +.
  3. When we want to call operator overloading function the syntax is:

     `operator sign object_name;`

     For example: For calling + sign's operator overloaded function we will use, `+a1;`

     Here, a1 is the object of a class where operator overloading function is declared.

4.10.1 Rules for Overloading Operators [S-18, 19]

- Although it looks simple to redefine the operators, there are certain restrictions and/or limitations in overloading them. Some of them are listed below:
  1. Only existing operators can be overloaded. New operators cannot be created.
  2. The overloaded operator must have at least one operand that is of user defined type.
  3. We cannot change the basic meaning of an operator. That is we cannot redefine the plus (+) operator to subtract one value from the other.
  4. Overloaded operators follow the syntax rules of the original operators that cannot be overridden.
  5. There are some operators that cannot be overloaded such as .*,:: etc.
  6. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values. But those overloaded by means of a friend function take one reference argument.

7. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
8. When using binary operators overloaded through a member function, the left-hand operand must be an object of the relevant class.
9. Binary arithmetic operators such as +, −, *, / must explicitly return a value. They must not attempt to change their own arguments.
10. We cannot use friend functions to overload certain operators, which are listed below. However, member functions can be used to overload them.

    = Assignment operator
    ( ) Function call operator
    [ ] Subscripting operator
    −> Class member access operator.

    Where, a friend cannot be used.

4.11 OVERLOADING UNARY AND BINARY OPERATORS

1. Overloading Unary Operator:

- The operators can be overloaded using two different functions i.e. through member functions and friend functions.
- A unary operator overloaded using a member function takes no argument, whereas an overloaded unary operator declared as friend function takes one argument.
- The unary operators that can be overloaded are shown below.

Table 4.2: Unary Operator

Operators	Meaning
−>	Indirect member operator
!	Logical negation
*	Pointer reference
&	Address operator
~	Ones complement
−> *	Indirect pointer to member
+	Addition
−	Subtraction
+ +	Incrementer
− −	Decrementer
−	Unary minus.

- First we will consider the unary minus (−) operator. A minus operator, when used as a unary, takes just one operand.
- We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an int or float variable.
- The unary minus when applied to an object should change the sign of each of its data items.

Program 4.20: Program for how the unary minus operator is overloaded. [S-18]

```
#include<iostream.h>
class space
{
    int x;
    int y;
    int z;
    public:
        void getdata (int a, int b, int c);
        void display (void);
        void operator - ( );  // overload unary minus
};
void space:: getdata (int a, int b, int c)
{
    x = a;
    y = b;
    z = c;
}
void space:: display (void)
{
    cout << x << " ";
    cout << y << " ";
    cout << z << "\n";
}
void space:: operator - ( ) // defining operator - ( )
{
    x = - x;
    y = - y;
    z = - z;
}
int main ( )
{  space s;
   s.getdata (10, -20, 30);
   cout << "s:  before overloading:";
   s.display ( );
   - s;                    // activates operator - ( )
   cout << "s: After overloading:";
   s.display ( );
}
```

Output:

```
s: before overloading:10 -20 30
s: After overloading:-10 20 -30
```

- Note that, the function operator – ( ) takes no argument. Then what does this operator function do? It changes the sign of data members of the objects.
- Since, this function is a member function of the same class; it can directly access the members of the object which activated it.
- Remember a statement like:

```
s2 = - s1;
```

Will not work because, the function operator – ( ) does not return any value.

- It can work if the function is modified to return an object.
- It is possible to overload a unary minus operator using a friend function as follows:

```
friend void operator - (space & s);    // declaration
void operator - (space & S)    // definition
{
    s.x = -s.x;
    s.y = -s.y;
    s.z = -s.z;
}
```

- Note that, the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator – ( ).
- Therefore, the changes made inside the operator function will not reflect in the called object.

Overloading of Increment Operator:

- We know that C++ supports the operator that is used for incrementing and decrementing by 1. These operators can be used either prefix or postfix.
- Generally, overloading of these operators cannot be distinguished between prefix or postfix operation. But whenever a postfix operation is overloaded, it takes a single argument along with a member function of a class object.

Program 4.21: Program to generate Fibonacci series by overloading a prefix operator.

```
#include<iostream.h>
class fibo
{
    private:
        int fib0, fib1, fib2;
    public:
        fibo(); // default constructor
        void operator ++();  // overloading declaration
        void disp();
};
```

```
fibo:: fibo()
{
    fib0 = 0;
    fib1 = 1;
    fib2 = fib0 + fib1;
}
void fibo:: disp()
{
cout<<fib2<<"\t";
 }
void fibo:: operator ++()
{
    fib0 = fib1;
    fib1 = fib2;
    fib2 = fib0 + fib1;
}
int main()
{
    fibo ob1;
    int n;
    cout<<"How many numbers you want?:";
    cin>>n;
    for(int i=0;i<=n;++i)
    {
        ob1.disp();
        ++ob1;
    }
}
```

Output:

```
How many numbers you want?:
4
1       2       3       5       8
```

2. Overloading Binary Operator:

- Binary operators are operator which requires two operands to perform operation.
- We have just seen how to overload a unary operator. The same mechanism can be used to overload a binary operator.
- To add two numbers generally we use statement like:

```
C = sum (A, B)     // functional notation
```

- The functional notation can be replaced by a natural looking expression,

```
C = A + B                    // arithmetic notation
```

By overloading the + operator using an operator + ( ) function.

- The binary operators overloaded through member function take one argument which is formal.
- The Table 4.3 shows the binary operators which can be overloaded. The binary overloaded operator function takes the first object as an implicit operand and the second operand must be passed explicitly.
- **Syntax for overloading a binary operator is:**

```
                keyword                 operator to be overloaded
                   ↑                        ↑
Return_type operator operator_symbol (arg)
                                          argument to operator function
{
   // body of operator function
}
```

Table 4.3: Binary Operators

Operator	Meaning
[ ]	Array element reference
( )	Function call
new	New operator
delete	Delete operator
*	Multiplication
/	Division
%	Modulus
+	Addition
–	Subtraction
<<	Left shift
>>	Right shift
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

contd. ...

==	Equal
!=	Not equal
&	Bitwise AND
^	Bitwise XOR
&&	Logical AND
||	Logical OR
=	Assignment
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign
+=	Add and assign
-=	Subtract and assign
<<=	Shift left and assign
>>=	Shift right and assign
&=	Bitwise AND and assign
:=	Bitwise OR and assign
^=	Bitwise one's complement and assign

Program 4.22: Program for binary operator overloading.

```
#include<iostream>
using namespace std;
class integer
{
   private:
      int val;
   public:
      integer(); //constructor1
      integer(int one); //constructor2
      integer operator + (integer objb); //operator function
      void disp();
};
integer:: integer()
{
   val = 0;
}
```

```
integer::integer(int one)
{
   val = one;
}
integer integer::operator + (integer objb)
{
   integer objsum;
   objsum.val = val + objb.val;
   return(objsum);
}
void integer::disp()
{
   cout<<"value="<<val<<endl;
}
int main()
{
   integer obj1 (11);
   integer obj2 (22);
    integer objsum;
   objsum = obj1 + obj2; //operator overloading
   obj1.disp();
   obj2.disp();
   objsum.disp();

}
```

Output:

```
value = 11
value = 22
value = 33
```

4.12 OPERATOR OVERLOADING USING FRIEND FUNCTION

- We have already seen that private members of a class cannot be accessed through outside functions. This is possible just by using friend function concept.
- Friend functions may be used in place of member functions for overloading a binary operator. The difference is friend function requires two arguments to be explicitly passed to it while a member function requires only one.

Overloading Binary Operators using 'friend' Functions:

- The syntax for declaring and defining a friend operator function to overload binary operator is as follows:

```
class class_name
{
    public:
        friend return_type operator op(arg1, arg2);
};
return_type operator op (arg1, arg2, … argN)
{
    //function definition ……..
}
```

Program 4.23: Program for overloading binary operator using friend function.

```
#include<iostream.h>
class binopr
{
    private:
        int x;
    public:
        binopr() { } // constructor
        binopr (int); // parameterized constructor
        void disp();
        friend binopr operator + (binopr,binopr);
};
binopr:: binopr(int a)
{
    x=a;
}
void binopr:: disp()
{
    cout<<x;
}
// overloading definition with friend function
binopr operator + (binopr P, binopr S)
{
    return (P.x + S.x);
}
```

```
int main()
{
    binopr ob1, ob2, ob3;
    ob1 = binopr (15);
    ob2 = binopr (28);
    ob3 = ob1 + ob2;
    ob3.disp();
}
```

Output:

```
43
43
Press any key to continue . . .
```

- The overloading definition can also be written using following code:

```
binopr operator + (binopr P, binopr S)
{
    S.x = P.x + S.x;
    return(S.x);
}
```

- The implementation of overloading + operator (binary) is shown in Fig. 4.12.

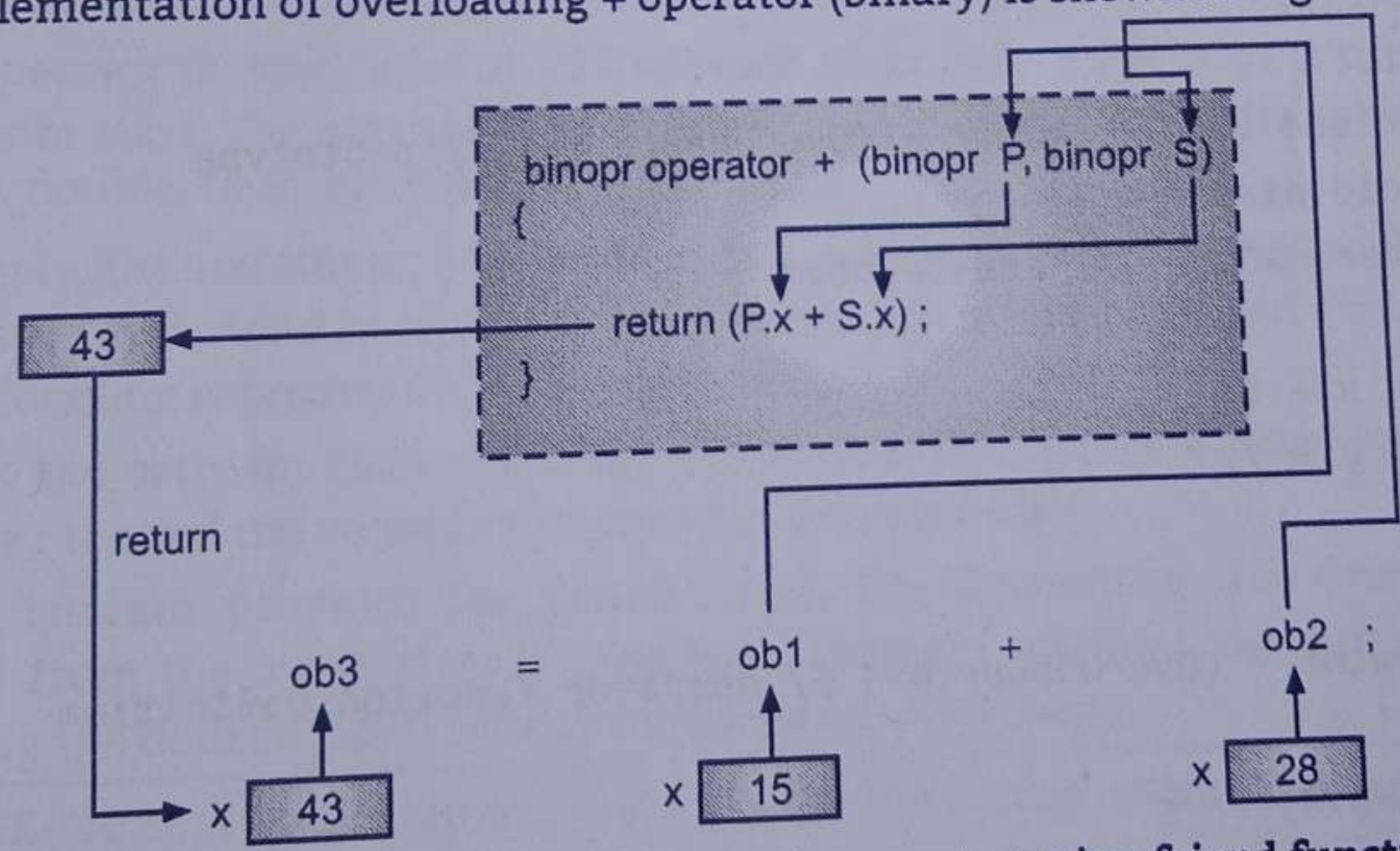


Fig. 4.12: Implementation of overloaded + operator using friend function

Overloading Unary Operators using 'friend' Functions:

- Here, the operator function is not the member function of any class. Hence, in this scenario if an operator has to be overloaded to make it work on a class object, the operator function has to be declared as a friend to that class.
- It is to be noted that, when an operator function is friend to a class, then we can call that function as a "friend operator function."

- The syntax for declaring and defining a friend operator function to overload an unary operator is as follows:

```
class class_name
{
    public:
        friend return_type operator op(arg);
};
return_type operator op(arg)
{
    //function definition ......
}
```

Program 4.24: Program for overloading unary operators using friend function.

```
#include<iostream>
using namespace std;
class OverUnaOpr
{
    private:
        int x;
        int y;
    public:
        friend void operator - (OverUnaOpr &); // prototype
        void display( );
        OverUnaOpr( )// constructor
        {
            x = 6;
            y = 7;
        }
};
void operator - (OverUnaOpr &a) // operator function definition
{
    a.x = -a.x;
    a.y = -a.y;
}
void OverUnaOpr:: display ( )
{
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
}
```

```
int main()
{
    OverUnaOpr s;
    s.display();
    -s;             //Overloading unary '-' operator
    s.display();
    return 0;
}
```

Output:

```
x = 6
y = 7
x = - 6
y = - 7
```

4.13 OVERLOADING INSERTION (<<) AND EXTRACTION (>>) OPERATORS [S-19, W-18]

- We have already used the objects **cin** and **cout** (pre-defined in iostream file) for the input and output of data of various types. This is possible only because the operators >> and << are overloaded to recognize all the basic C++ types.
- The >> operator is overloaded in the **istream** class and << operator is overloaded in the **ostream** class. The istream class overloads the >> operator for the standard types [int, long, double, float, char, and char * (string)].
- For example, the statement, `cin >> x;` calls the appropriate >> operator function for the istream cin defined in iostream.h. And uses it to direct this input stream into the memory location represented by the variable x.
- Similarly, the ostream class overloads the << operator, which allows the statement, `cout << x;` to send the value of x to ostream cout for output.
- Actually, istream provides the generic code for formatting the data after it is extracted from the input stream. Similarly, ostream provides the generic code for formatting the data before it is inserted to the output stream.

Program 4.25: Accept and display the information using extraction and insertion operators.

```
#include<iostream>
using namespace std;
class info
{
    private:
        int roll;
        char name[20];
```

```
    public:
        info()
        {
            roll = 0;
            name [0] = '0';
        }
        friend istream & operator>>(istream &, info&);
        friend ostream & operator<<(ostream &, info&);
};
istream & operator>>(istream &S, info &d)
{
    cout<<"Enter Rollno";
    S>>d.roll;
    cout<<"Enter name";
    S>>d.name;
    return S;
}
ostream & operator<<(ostream &S, info &d)
{
    S<<d.roll;
    S<<d.name;
    return S;
}
int main ()
{
    info ob1;
    cin>>ob1;
    cout<<ob1;
}
```

Output:

```
Enter rollno: 101
Enter name: Prajakta
101 Prajakta.
```

4.14 USAGE OF 'this' POINTER

- Every object in C++ has access to its own address through an important pointer called 'this' pointer.
- In other words, "The member functions of each and every object have access to a pointer named 'this', which points to the object itself.
- The 'this' pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

- Friend functions do not have a 'this' pointer, because friends are not members of a class. Only member functions have a 'this' pointer.
- 'this' is a pointer that points to the object for which the function was called.
- 'this' pointer is predefined pointer variable within every class.
- 'this' pointer is a pointer pointing to the current object of specified class. "this" is a keyword in C++.
- **Syntax:**

```
class_name * this;
```

- Every object has access to its own address through a pointer called **this**. When a member function is called, 'this' pointer is automatically passed as an implicit argument to that function.
- For example: The function call,

  `s.sample();` will set the pointer this to the address of the object s.
- One of the important applications of "this" pointer is to return the object to which it is pointing. For example the following statement,

```
return * this;
```

Inside a member function will return the object that invokes a member function.

Program 4.26: Write a program to class test having data members as name of a student and marks. Accept and display information by using 'this' pointer.

```
#include<iostream>
using namespace std;
class test
{
   private:
      char name[40];
      float marks;
   public:
   void get()
   {
      cout << "\n Enter name & marks";
      cin >> name >> marks;
   }
   void display()
   {
      this -> get();
      cout << "\n Name =" << this -> name;
      cout << "\n Marks = "<< this -> marks;
   }
};
```

```
int main()
{
    test t1, t2;
    t1.display();
    t2.display();
    return 0;
}
```

Output:

```
Enter name & marks
Prajakta 80

Name =Prajakta
Marks = 80
```

4.15 TYPE CONVERSION

- Type conversion is also known as **Implicit Type Conversion**. It is done by the compiler on its own. There is no external trigger required by the user to typecast a variable from one type to another.
- This occurs when an expression contains variables of more than one type. So, in those scenarios automatic type conversion takes place to avoid loss of data. In automatic type conversion, all the data types present in the expression are converted to data type of the variable with the largest data type.
- For example:

```
short a=2000;
int b;
b=a;
```

- Here, the value of a has been promoted from short to int and we have not had to specify any typecasting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.

Program 4.27: C++ program for type conversion.

```
#include <iostream>
using namespace std;
int main()
{
  int a = 9;
  double b;
  b = a;
```

```
    cout << "int = " << a << endl;
    cout << "double = " << b << endl;
    return 0;
  }
```

Output:

```
int = 9
double = 9
```

4.16 FUNCTION OVERLOADING

This point is already covered in same unit (See point 4.7)

Summary

- Operator overloading refers to overloading of one operator for many different purpose.
- Operator overloading is a compile time polymorphism in which a single operator is overloaded to give user defined meaning to it. Operator overloading provides a flexibility option for creating new definitions of C++ operators.
- Operator overloading is done using operator functions. Operator functions can be a member function or a friend function.
- Operator overloading is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.
- Operator overloading mainly comes into picture when the C++ operators are being operated on class objects.
- The member functions of each and every object have access to a pointer named 'this', which points to the object itself.
- The 'this' pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.
- Constructor is a special member function which is having same name as class name which is used to automatically initialize data members to some values when an object of the class is created.
- Constructor is a special member function of the class. A constructor's task is to allocate the memory and initialize the objects of its class. It is special because its name is same as the class name.
- The constructor is invoked whenever; an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.
- A constructor which does not contain any parameters is called a default constructor or also called as no parameter constructor or empty constructor. It is used to initialize the data members to some default values.
- A constructor which accepts one or more parameters is called a parameterized constructor.
- A constructor which accepts an already existing object through reference to copy the data member values is called a copy constructor. As the name implies a copy constructor is used to create a copy of an already existing object.

- Constructor overloading is as similar as function overloading where we can have multiple functions.
- A constructor that accepts default arguments is called as default argument constructor.
- A constructor that allocates memory to the objects at the time of their construction using the new operator is known as dynamic constructor.
- A destructor is a special member function of a class that is executed whenever an object of its class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.
- A destructor will have exact same name as the class prefixed with a tilde (~) sign and it can neither return a value nor can it take any parameters.
- Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Check Your Understanding

1. Constructors are used to _____ the object.
   (a) increment (b) initialize
   (c) destroy (d) both (a) & (b)
2. Destructors are used to _____ the object.
   (a) increment (b) initialize
   (c) destroy (d) modify
3. Name of constructor is same as the _____.
   (a) class (b) object
   (c) member function (d) both (a) & (b)
4. The memory is allocated to the data members of class, when _____ of the class is declared.
   (a) constructor (b) destructor
   (c) both (a) & (b) (d) object
5. In operator overloading _____ in the basic meaning of operator is not allowed.
   (a) increment (b) addition
   (c) change (d) none
6. If we overload only prefix ++ operator, the postfix ++ operator _____.
   (a) does not work (b) works naturally
   (c) works arbitrarily (d) works as if prefix ++ operator
7. When the compiler decides binding as overload member, it is called _____ binding.
   (a) local (b) static
   (c) dynamic (d) safe
8. Which of the following operator cannot be overloaded?
   (a) + (b) *
   (c) :: (d) both (a) and (b)

9. Which of the following is true about this pointer?
   (a) It is passed as a hidden argument to all function calls.
   (b) It is passed as a hidden argument to all non-static function calls.
   (c) It is passed as a hidden argument to all static functions.
   (d) None of the above.

Answers

1. (b)	2. (c)	3. (a)	4. (d)	5. (c)	6. (d)	7. (b)	8. (c)	9. (b)

Practice Questions

Q.I Answer the following questions in short:

1. What are constructor and destructor?
2. How many times will the constructor of class student are invoked for the following statement. student S, * P;
3. State the difference between constructor and destructor.
4. What is destructor? How many destructors can be defined in a single class?
5. Give any four characteristics of constructor.
6. What are the ways of constructor calling in main program? Give their syntax.
7. What is mean by copy constructors?
8. What is Operator Overloading?
9. What is meant by 'this' Pointer?
10. Explain rules for operating overloading.
11. What is function overloading?

Q.II Answer the following questions:

1. Write short note on: Default constructor.
2. Explain multiple constructor with example.
3. What is static data member? Where to declare them?
4. What is meant by default argument? Illustrate concept of constructor with default argument with suitable example.
5. Define parameterized constructor with its syntax and example.
6. Declare a class simple interest having data members as principle amount rate of interest, number of years. The constructor will have default value of rate of interest as 11.5%. Accept this data for two objects. Calculate and display simple interest for each object.
7. Write a program to declare class time having data members as hrs, min. sec. Write a constructor to accept data and use display function for two objects.
8. What are the rules for writing destruction function and when they are invoked?
9. Write a program to find sum of nos. between 1 to n using constructor where value of n will be passed to the constructor.

10. Write a program in C++ which prints the factorial of a given number using a constructor and destructor member function.
11. Write a program in C++ which prints the factorial of a given number using a copy constructor and a destructor member function.
12. Write a program in C++ that determines whether a given number is prime or not and print that number using default constructor and destructor member function.
13. State any four differences between constructor and destructor.
14. Why it is necessary to overload an Operator?
15. Explain the rules for Overloading Operators.
16. Explain the usage of 'this' Pointer.
17. What are the C++ operators that can be used for binary and unary applications?
18. Write a C++ program for a class integer which contains an integer as a data member overload the ! operator to find the factorial of an integral.

Q.III Define terms:

1. Copy constructor
2. Constructor
3. Parameterized constructor
4. Dynamic constructor
5. Overloaded constructor.
6. this pointer
7. Operator overloading

Previous Exam Questions

Summer 2018

1. What are mandatory parts in function declarations? **[1 M]**

(i) return type, function name
(ii) return type, fun name, parameter
(iii) both (i) & (ii)
(iv) none of these

Ans. (i) return type, function name

2. Write the operators which cannot be overloaded. **[1 M]**

Ans. Refer to Section 4.10.

3. What is dynamic constructor? **[1 M]**

Ans. Refer to Section 4.8.3.4.

4. Explain function overloading with example. **[5 M]**

Ans. Refer to Section 4.7.

5. What is friend function? Explain with properties & example. **[4 M]**

Ans. Refer to Section 4.6.

6. Write a C++ program to find maximum no of three number using parameterized constructor. **[3 M]**

Ans. Refer to Program 4.13.

7. Write C++ program for how the unary minus operator is overloaded. [5 M]

Ans. Refer to Program 4.20.

8. Write the rules of operator overloading. [3 M]

Ans. Refer to Section 4.10.1.

Winter 2018

1. A _____ function is a non-member function of a class but can access private data members.
   (a) inline
   (b) static
   (c) private
   (d) friend

Ans. (d) friend

2. Whenever an object is destroyed _____ function is called.
   (a) destructor
   (b) constructor
   (c) copy constructor
   (d) default constructor

Ans. (a) destructor

3. Default constructor is always called without any arguments. State true or false. [1 M]

Ans. True

4. Define constructor. Explain any two types of constructor. [5 M]

Ans. Refer to Section 4.8.

5. What is friend function? Explain with the help of an example. [4 M]

Ans. Refer to Section 4.6.

6. What is copy constructor? Explain with the help of an example. [4 M]

Ans. Refer to Section 4.8.3.3.

7. Trace the output of the following code (Assume there are no syntax errors): [3 M]

```
Class A
{
public :
A( )
{
count <<"\n object created";
}
~ A( )
{
cout <<"\n object destroyed";
}
};
```

```
A a1;
main( )
{
Aa2;
{
Aa3;
}
exit(0);
}
```

9. What is operator overloading? Write the rules for operator overloading. **[3 M]**

Ans. Refer to Section 4.10.

10. What is function overloading? Explain with the help of an example. **[4 M]**

Ans. Refer to Section 4.7.

11. Write the syntax for overloading insertion and extraction operator. **[3 M]**

Ans. Refer to Section 4.13.

Summer 2019

1. Function overloading means defining multiple functions having _____ name, which performs the _____ logical task.

(a) same, different
(b) same, same
(c) different, same
(d) different, different

Ans. (a) same, different

2. Define constructor. **[1 M]**

Ans. Refer to Section 4.8.

3. Give the syntax for overloading insertion and extraction operators. **[1 M]**

Ans. Refer to Section 4.13.

4. What is Friend Function? Give its features. **[4 M]**

Ans. Refer to Section 4.6.

5. Give difference between call by value and call by reference. **[3 M]**

6. State rules for operator overloading. **[3 M]**

Ans. Refer to Section 4.10.1.

7. Explain copy constructor with example. **[4 M]**

Ans. Refer to Section 4.8.3.3.

8. Give advantages and disadvantages of inline functions. **[3 M]**

Ans. Refer to Section 4.5.

5... Inheritance

Objectives...

- To study how to define Base class and Derived class.
- To learn about different Types of Inheritance.
- To understand Virtual Base Class.
- To study Abstract class.
- To learn Polymorphism.
- To study Virtual Function and Pure Virtual Function.

5.1 INTRODUCTION [S-19]

- Inheritance is a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behavior and overriding these with capabilities the new classes requires.
- Inheritance is the process by which one object can acquire the properties of another object. Inheritance is the process of creating new classes from an existing class.
- The existing class is known as base class and the newly created class is called as a derived class. The derived class inherits all capabilities of the base class. A derived class is more specific than its base class and represents a smaller group of objects.
- For example, a Red Delicious apple is part of the classification apple, which in turn is part of the fruit class. Hence, inheritance supports the concept of classification; with the help of classification, an object need to define only those qualities that make it unique within its class.

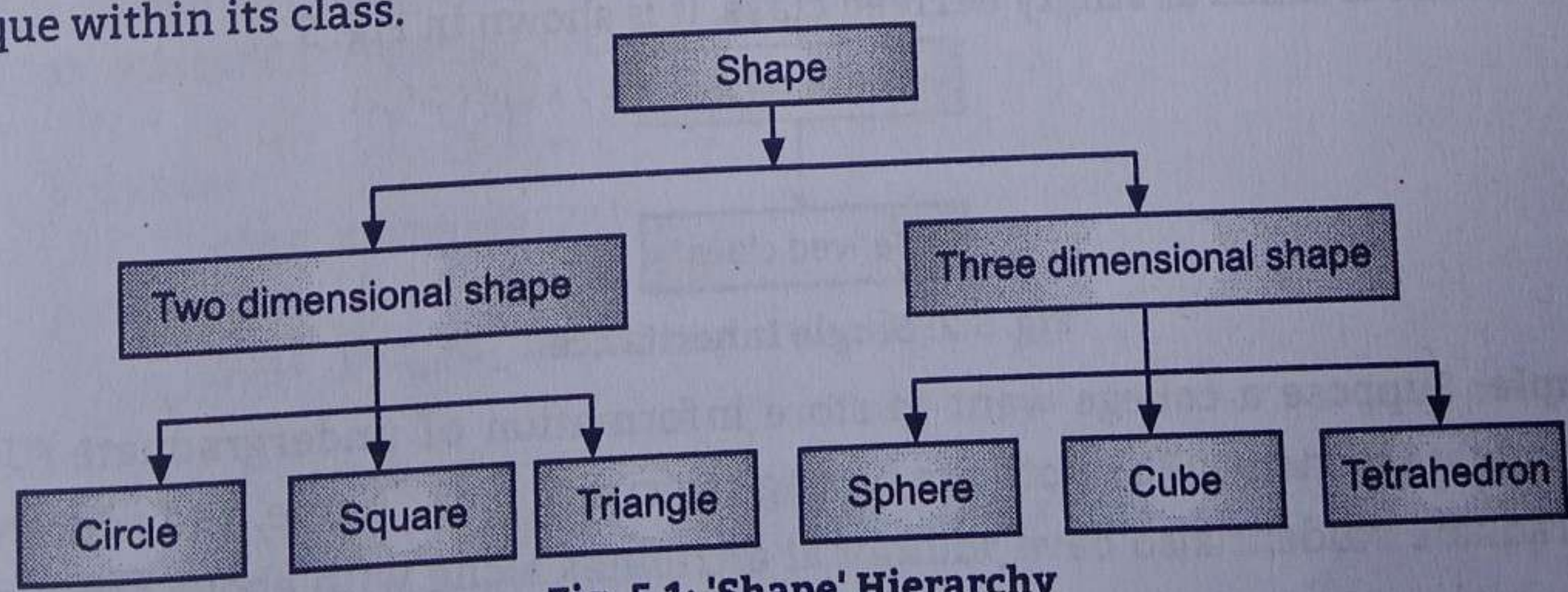


Fig. 5.1: 'Shape' Hierarchy

- The inheritance makes it possible for one object to be a specific instance of a more general case. Example of inheritance hierarchy of shape is shown in Fig. 5.1.
- The main advantages of inheritance are:
  1. It helps to implement software reusability. Once the hierarchy is done, the specific data to the particular class can be added as per the specification of the derived classes in hierarchy. Reusing existing code saves time and money.
  2. It increases the reliability of the code.
  3. Inheritance can also help in the original conceptualization of a programming problem and in the overall design of the program.
  4. It is useful to avoid redundancy, leading to smaller models that are easier to understand.
  5. It adds some enhancements to the base class.
  6. We can create new class based on base class without modifying base class.

5.2 TYPES OF INHERITANCE [S-18]

- There are many ways in which we can create derived class. There are five types of inheritance as follows:
  1. Single inheritance
  2. Multilevel inheritance
  3. Multiple inheritance
  4. Hierarchical inheritance and
  5. Hybrid inheritance

5.2.1 Single Inheritance [S-18]

- When one base class is derived by one child class, then it is known as single inheritance. The existing base class is known as direct base class whereas, they newly created class is called as **singly derived class**. It is shown in Fig. 5.2.

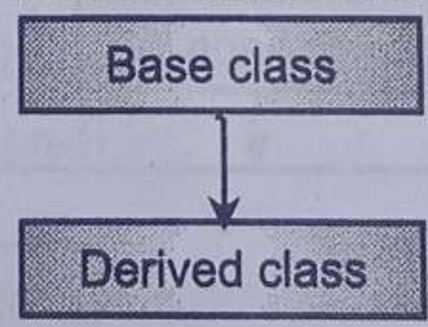


Fig. 5.2: Single Inheritance

- **Example:** Suppose a college want to store information of undergraduate (UG) and postgraduate students (PG). Both are having registration no., name, and address. Each postgraduate student also have additional attributes along with above attributes are placement company name. It can show as in Fig. 5.3.

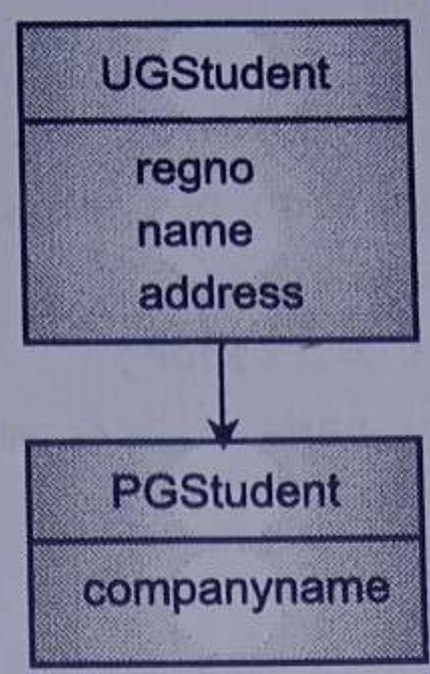


Fig. 5.3

Program 5.1: Program to illustrate single inheritance.

```
#include<iostream>
using namespace std;
class student
{
    protected:
        int regno;
        char name[20];
        char addr[30];
    public:
        void getdata()
        {   cout<<"enter registration no, name and address";
            cin>>regno>>name>>addr;
        }
        void display()
        {
            cout<<"registration no.= "<<regno;
            cout<<"name = "<<name;
            cout<<"address = "<<addr;
        }
};   //end of base class
class PGstudent: public student
{
    private:
        char compname[20];
    public:
        void get_PG_data()
        {
            getdata();
            cout<<"\nEnter the company name=";
            cin>>compname;
        }
```

```
        void dispPG()
        {
            cout<<"\n registration no. ="<<regno;
            cout<<"\n name ="<<name;
            cout<<"\n address ="<<addr;
            cout<<"\n company name = "<<compname;
        }
};
int main()
{
    student s;
    PGstudent ps;
    s.getdata();
    s.display();
    ps.get_PG_data();
    ps.dispPG();
}
```

Output:

```
Enter registration no, name and address
101 Ram Pune
registration no = 101  name = Ram      addr = Pune
Enter registration no., name & address
401 Sita Pune
Enter the company name = zenex
Registration no = 401
name = Sita
address = Pune
Company name = Zenex
```

Program 5.2: A base class contains the data of employee name, empno and sex. The derived class contains basic salary. The derived class has been declared as an array of class objects. The program illustrate the single inheritance.

```
#include<iostream.h>
class base
{
    private:
        char name[20];
        int empno;
        char sex;
```

```
    public:
        void get();
        void disp();
};
class derived: public base
{
    private:
        float bsalary;
    public:
        void get();
        void disp();
};
void base:: get()
{
    cout<<"\n Enter name:";
    cin>>name;
    cout<<"\n Enter Employee id:";
    cin>>empno;
    cout<<"\n Enter Sex:";
    cin>>sex;
}
void base:: disp()
{
    cout<<"name ="<<name;
    cout<<"Emp no. is ="<<empno;
    cout<<"sex="<<sex;
}
void derived:: get()
{
    base:: get();
    cout<<"Enter basic salary: \n";
    cin>>bsalary;
}
void derived:: disp()
{
    base::disp();
    cout<<"basic salary = "<<bsalary;
}
```

```
void main()
{
    derived ob[10];
    int i, n;
    cout<<"How many employees?";
    cin>>n;
    for (i=0;i<n;i++)
    {
        ob[i].get();
    }
    cout<<endl;
    cout<<"name Empno.sex bsalary \n";
    for (i=0;i<n;i++)
    {
        ob[i].disp();
        cout<<endl;
    }
}
```

Output:

```
How many employees? 2
Enter name: Prajakta
Enter Employee id: 101
Enter Sex: f
Enter basic salary: 10000
Enter name: Pranjal
Enter Employee id: 102
Enter Sex: m
Enter basic salary: 40000

name        Empno        sex        bsalary
name =abc   Emp no. is =101   sex=f   basic salary = 10000
name =xyz   Emp no. is =102   sex=m   basic salary = 40000
```

5.2.2 Multilevel Inheritance

- When a base class is derived by a child class which further derived by another child class, then it is known as Multilevel Inheritance.
- The class which provides link between two classes is known as intermediate base class. This is illustrate in Fig. 5.4.

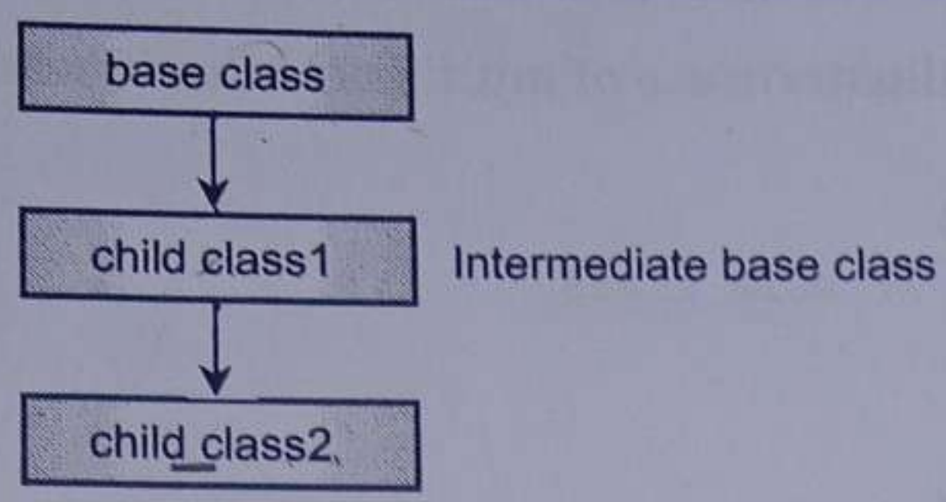


Fig. 5.4: Multilevel Inheritance

Practically, we can have more than two levels also.

- **Syntax of multilevel inheritance:**

```
class base
{
    ........
    ........
};
class child1: public base
{
    ........
    ........
};
class child2: public child1
{
    ........
    ........
};
```

- Consider an example of bank. In a bank, different customers have savings account. Some customers may have taken loan from the bank. So the bank always maintains information about bank depositors and borrowers. This can be shown using the following Fig. 5.5.

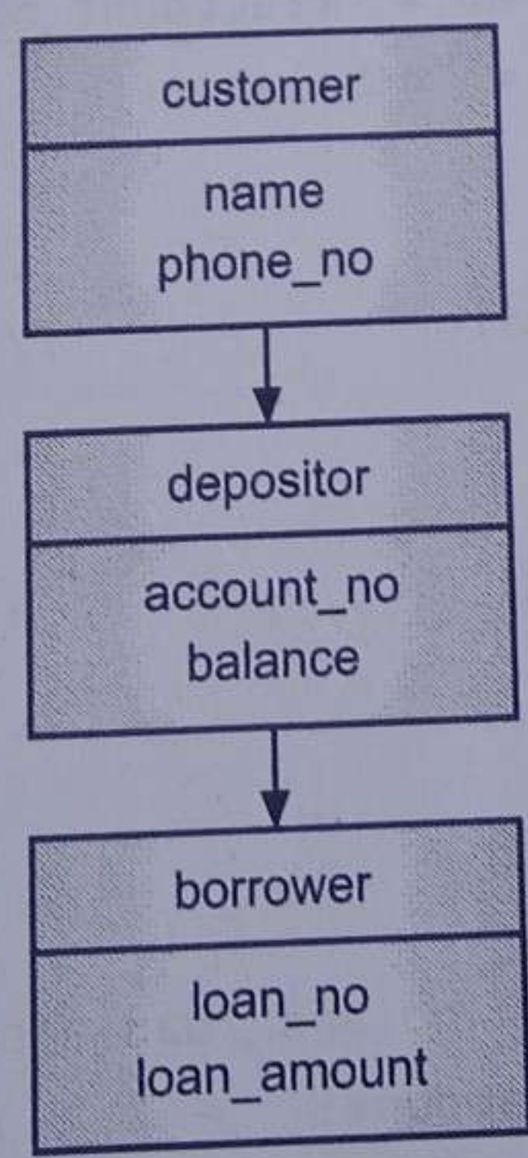


Fig. 5.5: Example of Bank

Program 5.3: Program to illustrate use of multilevel inheritance.

```
#include<iostream>
using namespace std;
class customer        // base class
{
   protected:
      char name[30];
      int phone_no;
}; // end of base class
class depositor: public customer     // child class1
{
   private:
      int account_no;
      float balance;
   public:
      void get_depositor_data( )
      {
         cout<<"\n Enter name & phone no ";
         cin>>name>>phone_no;
         cout<<"\n Enter account no and balance ";
         cin>>account_no>>balance;
      }
      void display_depositor_data( )
      {
         cout<<"\n Name = "<<name;
         cout<<"\n Phone no = "<<phone_no;
         cout<<"\n Account no = "<<account_no;
         cout<<"\n Balance = "<<balance;
      }
}; // end of child class1
class borrower: public depositor              // child class2
{
   protected:
      int loan_no;
      float loan_amount;
   public:
      void get_loan_data( )
      {
         cout<<"\n Enter loan no and loan amount";
         cin>>loan_no>>loan_amount;
      }
```

```
        void display_loan_data( )
        {
            cout<<"\n Loan no = "<<loan_no;
            cout<<"\n Loan amount = "<<loan_amount;
        }
};  // end of child class2
int main( )
{
    int choice;
    cout<<"1-Read & display depositor information "<<endl;
    cout<<"2-Read & display depositor & borrower information"<<endl;
    cout<<"Enter your choice = ";
    cin>>choice;
    switch (choice)
    {
        case 1:
            depositor d;
            d.get_depositor_data( );
            d.display_depositor_data( );
            break;
        case 2:
            borrower b;
            b.get_depositor_data( );
            b.get_loan_data( );
            b.display_depositor_data( );
            b.display_loan_data( );
            break;
    }
}  // end of main( )
```

Output:

```
1-Read & display depositor information
2-Read & display depositor & borrower information
Enter your choice = 1
Enter name & phone no Prajakta 98776677
Enter account no and balance 123456789 10000
Name = Prajakta
Phone no = 98776677
Account no = 123456789
Balance = 10000
```

- In the above program, though the data members of depositor class are private, they can be accessed indirectly by the borrower class through the member functions of the depositor class indirectly.

Program 5.4: Consider the example of student data. Base class is student. Class test is intermediate base class and result is child class. The class result inherits the details of marks obtained in the test and the roll no. of student from base class through multilevel inheritance.

```
#include<iostream>
using namespace std;
class student
{
   protected:
      int roll;
   public:
      void get(int x)
      {
         roll = x;
      }
      void display()
      {
         cout<<"roll="<<roll;
      }
};
class test: public student
{
   protected:
      int mark1, mark2;
   public:
      void getmark(int a, int b)
      {
         mark1 = a;
         mark2 = b;
      }
      void dispmark()
      {
         cout<<"mark1 = \n"<<mark1<<"mark2 ="<<mark2;
      }
};
```

```
class result: public test
{
    private:
        int total;
    public:
        void displayall()
        {
            total = mark1 + mark2;
            display();
            dispmark();
            cout<<"total="<<total;
        }
};
int main()
{
    result r;
    r.get(10);
    r.getmark(60,75);
    r.displayall();
}
```

Output:

```
roll = 10
mark1 = 60
mark2 = 75
total = 135
```

5.2.3 Multiple Inheritance

- When more than one base classes are inherited by a derived class, then such type of inheritance is called as multiple inheritance. Fig. 5.6 illustrates the point.

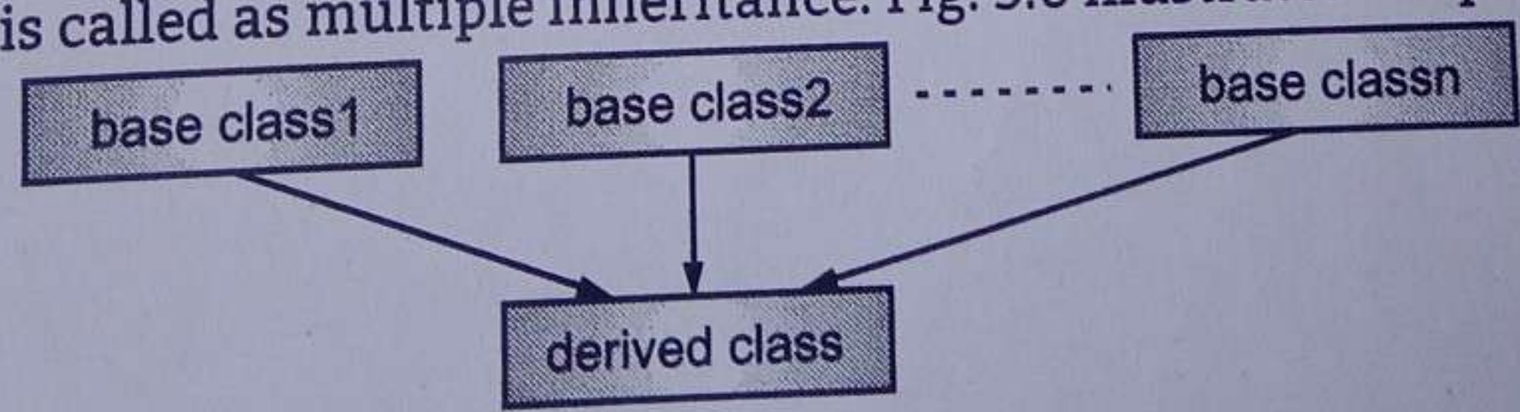


Fig. 5.6: Multiple Inheritance

- **Example:** A child inherits the features from father and intelligence from mother.

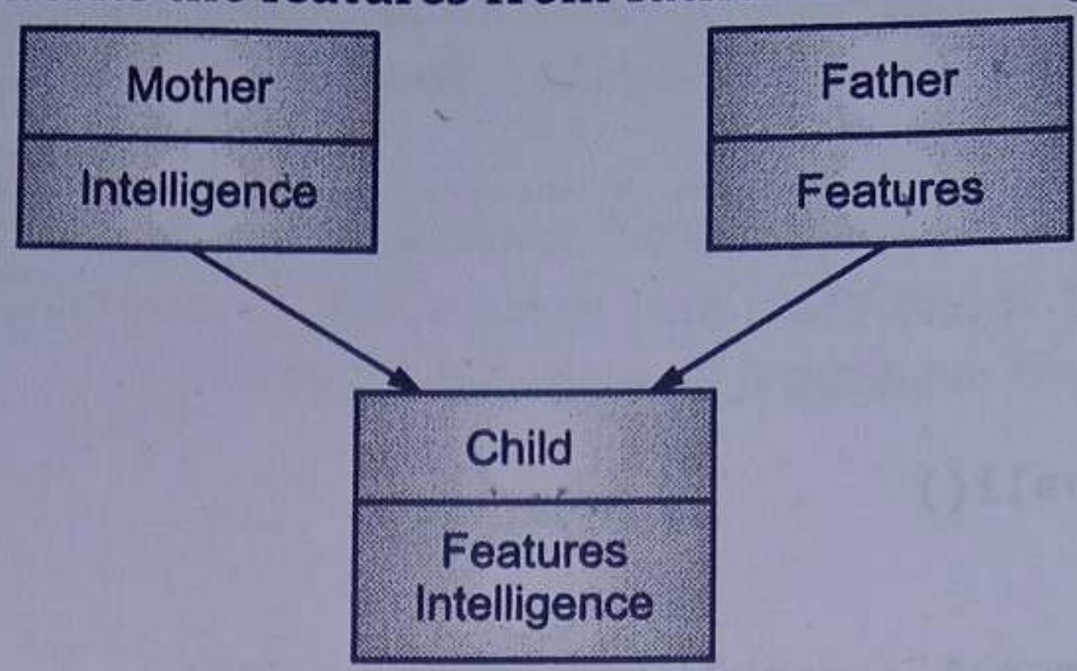


Fig. 5.7: Multiple Inheritance

- **Syntax of Multiple Inheritance:**

1. **Multiple inheritance with all public derivation:**

```
class base1
{
    ......
    ......
    ......
};
class base2
{
    ......
    ......
    ......
};
class derived: public base1, public base2
{
    ......
    ......
    ......
};
```

Here access specifier may be public or private and the base classes are separated by commas.

2. **Multiple inheritance with all private derivation:**

```
class base1
{
    ......
    ......
    ......
};
```

```
class base2
{
    ......
    ......
    ......
};
class derived: private base1, private base2
{
    ......
    ......
    ......
};
```

3. Multiple inheritance with all mixed derivation:

```
class base1
{
    ......
    ......
    ......
};
class base2
{
    ......
    ......
    ......
};
class base3
{
    ......
    ......
    ......
};
class derived: pubic base1, public base2, private base3
{
    ......
    ......
    ......
};
```

Let us discuss the program how multiple inheritance works.

Program 5.5: Program for multiple inheritance.

```
#include<iostream>
using namespace std;
class basex
{
    protected:
        int x;
    public:
        void get_x (int a) {x = a;}
};
class basey
{
    protected:
        int y;
    public:
        void get_y (int b) {y = b;}
};
class derived: public basex, public basey
{
    public:
        void display()
        {
            cout<<"x = " <<x<<"\n";
            cout<<"y = "<<y<<"\n";
            cout<<"x+y = "<<x+y<<"\n";
        }
};
int main()
{
    derived ob;
    ob.get_x(50);
    ob.get_y(60);
    ob.display();
}
```

Output:

```
x = 50
y = 60
x + y = 110
```

Program 5.6: Consider an example to print bio-data of a diploma student having personal and academic information. Program to illustrate use of multiple inheritance.

```
#include<iostream>
#include<cstring>
using namespace std;
class personal                                  // base class1
{
   private:
      char name [30], email_id[50];
   protected:
   void get_personal_info( )
   {
      cout<<"\n Enter name & email id of a person = ";
      cin>>name>>email_id;
   }
   void display_personal_info( )
   {
      cout<<"\n Name = "<<name;
      cout<<"\n Email_id = "<<email_id;
   }
};  // end of base class1
class academic                                  // base class 2
{
   private:
      float tenth_marks;
      char tenth_class[20];
   protected:
      void get_academic_info( )
      {
         cout<<"\n Enter tenth marks";
         cin>>tenth_marks;
      }
      void find_class( )
      {
         if (tenth_marks>=70)
            strcpy (tenth_class, "Distinction");
         else
         if (tenth_marks>=60)
            strcpy (tenth_class, "First class");
```

```
        else
        if (tenth_marks>=50)
            strcpy (tenth_class, "Second class");
        else
        if (tenth_marks>=40)
            strcpy (tenth_class, "Pass class");
        else
            strcpy (tenth_class, "Fail");
    }
    void display_academic_info( )
    {
        cout<<"\n Tenth marks = "<<tenth_marks;
        find_class( );
        cout<<"\n Tenth class = ";
        cout<<tenth_class;
    }
}; // end of base class2
class bio_data: private personal, private academic      // Derived class
{
    public:
        void display_biodata( );
        void get_info( );
    }; // end of derived class
        // Member functions of derived class
    void bio_data::display_biodata( )
    {
        cout<<"BIO DATA "<<endl;
        cout<<"Personal info = "<<endl;
        display_personal_info( );
        cout<<"Academic info= "<<endl;
        display_academic_info( );
    }
    void bio_data::get_info( )
    {
        get_personal_info( );
        get_academic_info( );
    }
    int main( )
    {
        bio_data b;
        // First read info
        b.get_info( );
        // Now print biodata
        b.display_biodata( );
    }
```

Output:

```
Enter name & email id of a person = Prajakta abc@yahoo
Enter tenth marks 80.5
BIO DATA
Personal info =
Name = Prajakta
Email_id = abc@yahoo
Academic info=
Tenth marks = 80.5
Tenth class = Distinction
```

Ambiguity in Multiple Inheritance: [W-18]

- One common cause of ambiguity in multiple inheritance is when two or more base classes have methods with the same name, while there is no method of that name in the derived class. In this case objects of the derived class have no way of knowing which of the parent methods is to be executed. The scope resolution operator can be used to resolve the ambiguity.
- Another element of ambiguity in multiple inheritance arises when two or more derived classes inherit the same member from a common base class. If now another derived class inherits from these classes, two copies of the original member function could be inherited.
- Fig. 5.8 shows one possible scenario that leads to inheritance of multiple copies.
- One possible solution to this duplicity is the use of the scope resolution operator. Another solution of this problem is to declare the base classes to be virtual at the time of they are inherited.

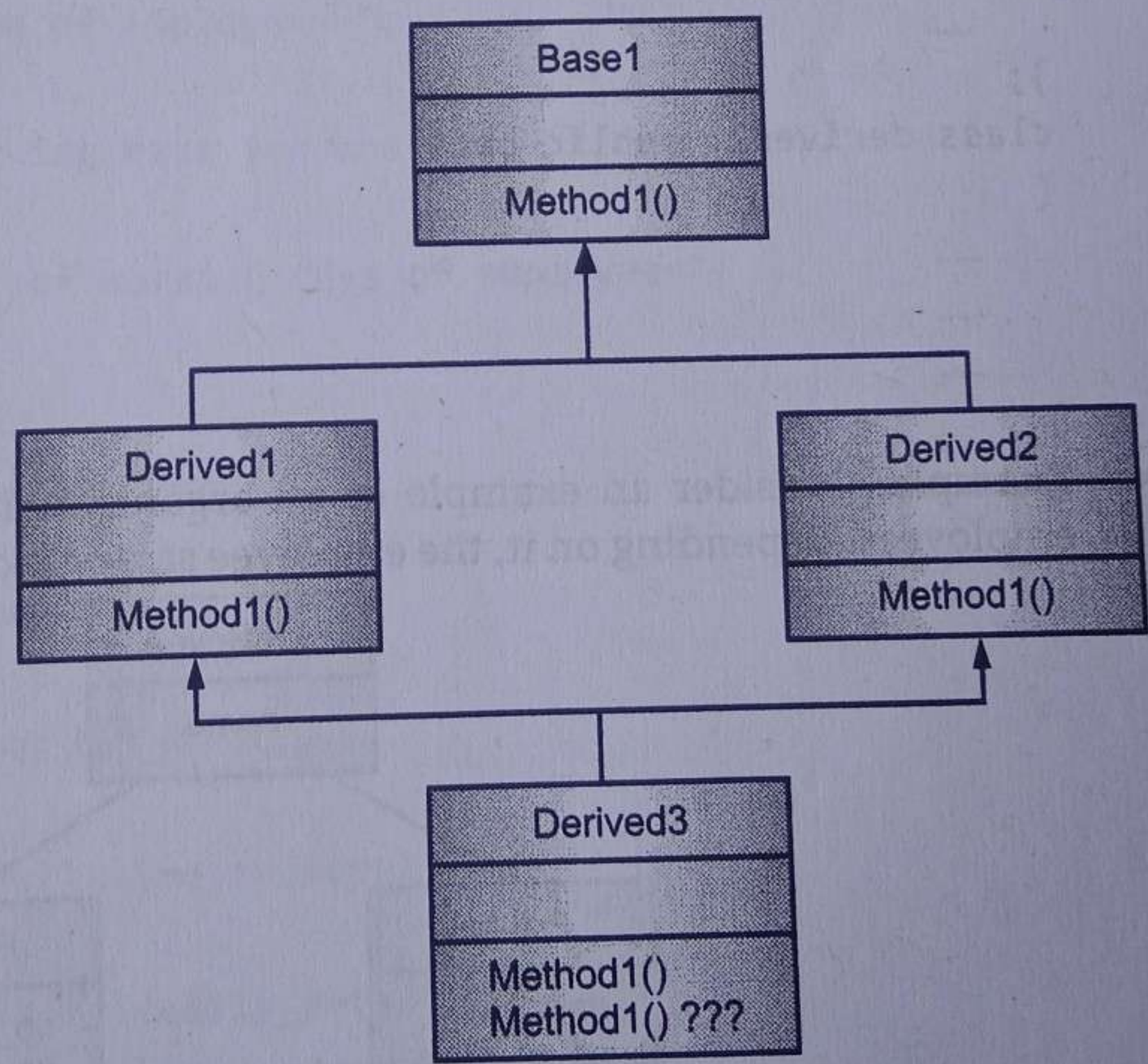


Fig. 5.8: Inheritance of Multiple Copies

Advantages of Multiple Inheritance:

1. Derived class typically represents a combination of its base classes.
2. It has rich semantics
3. It has ability to directly express complex structures.

5.2.4 Hierarchical Inheritance

- When one base class is inherited by more than one derived classes, it is known as hierarchical inheritance. In this, many programming problems can be cast into a hierarchy where certain features of one level are shared by many other below that level.

  The Fig. 5.9 illustrate this point.

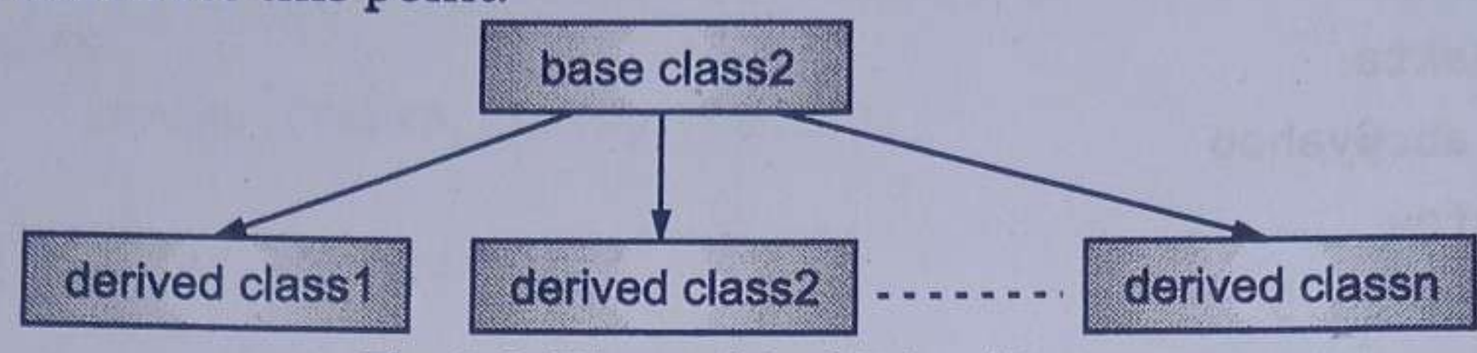


Fig. 5.9: Hierarchical Inheritance

- **Syntax of Hierarchical Inheritance:**

```
class base
{
    ......
    ......
    ......
};
class derived1: public base
{
    ......
    ......
    ......
};
class derived2: public base
{
    ......
    ......
    ......
};
```

- **Example:** Consider an example of an organisation having full time and part time employees. Depending on it, the employee salary is calculated. The hierarchy is:

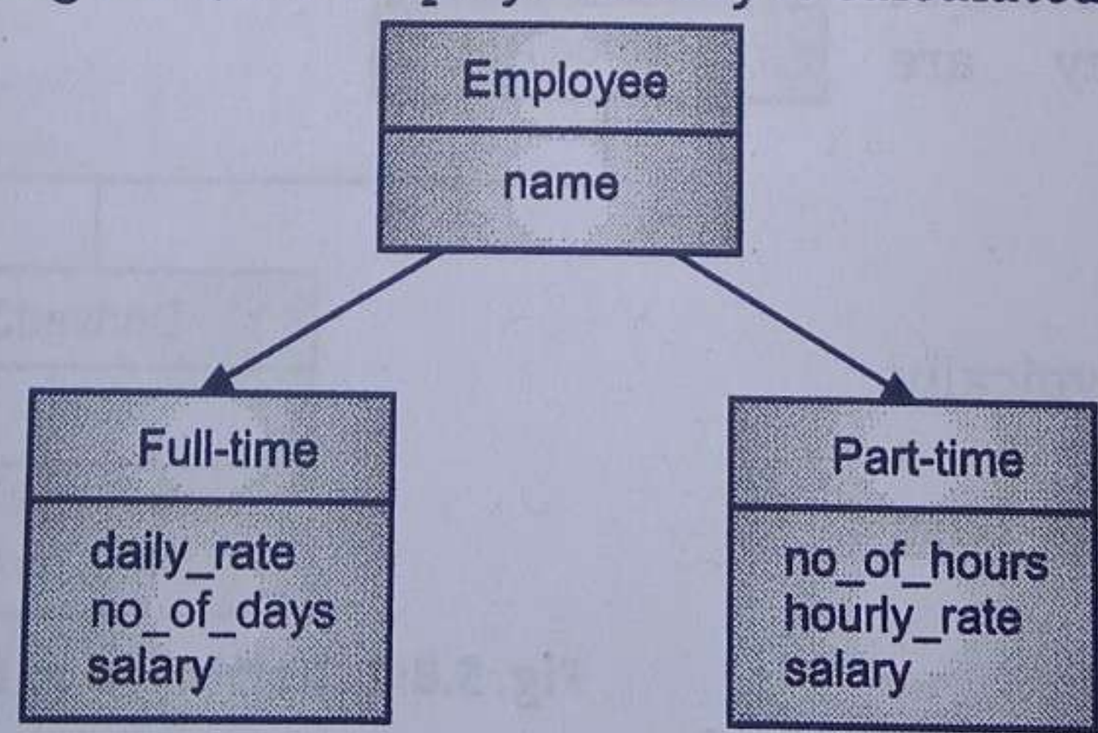


Fig. 5.10: Hierarchical Inheritance

Program 5.7: Program to illustrate use of hierarchical inheritance.

```
#include<iostream>
using namespace std;
class employee  // base class
{
   protected:
      char name[30];
}; // end of base class
class full_time_emp: protected employee
{
   protected:
      float daily_rate, salary;
      int no_of_days;
   public:
      void read_emp_data( );
      void display_emp_data( );
      void calculate_salary( );
}; // end of derived class1

void full_time_emp::read_emp_data( )
{
   cout<<endl<<"Enter name of employee =";
   cin>>name;
   cout<<endl<<"Enter working rate for one day = ";
   cin>>daily_rate;
   cout<<endl<<"Enter no. of working days of employee=";
   cin>>no_of_days;
}
void full_time_emp::calculate_salary( )
{
   salary = daily_rate *no_of_days;
}
void full_time_emp::display_emp_data( )
{
   cout<<endl<<"For the full time employee:=";
   cout<<endl<<"Name="<<name;
   cout<<endl<<"Daily rate ="<<daily_rate;
   cout<<endl<<"No. of working days ="<<no_of_days;
   cout<<endl<<"Salary ="<<salary;
}
```

```
class part_time_emp:protected employee      // derived class2
{
   protected:
         float hourly_rate, salary;
         int no_of_hours;
   public:
         void read_emp_data( );
         void display_emp_data( );
         void calculate_salary( );
 }; // end of derived class 2
 void part_time_emp::read_emp_data( )
 {
    cout<<endl<<"Enter name of employee=";
    cin>>name;
    cout<<endl<<"Enter working rate for one hour =";
    cin>>hourly_rate;
    cout<<endl<<"Enter no. of working hours for employee =";
    cin>> no_of_hours;
 }
 void part_time_emp::calculate_salary( )
 {
    salary = hourly_rate * no_of_hours;
 }
 void part_time_emp::display_emp_data( )
 {
    cout<<endl<<"For the part time employee";
    cout<<endl<<"Name ="<<name;
    cout<<endl<<"Hourly rate ="<<hourly_rate;
    cout<<endl<<"No. of working hours="<<no_of_hours;
    cout<<endl<<"Salary="<<salary;
 }
 int main( )
 {
    int choice;
    cout<<endl<<"You have following choices";
    cout<<endl<<"1-Read & display information of full time employee";
    cout<<endl<<"2-Read & display information of part time employee";
    cout<<endl<<"Enter your choice=";
    cin>>choice;
```

```
    switch(choice)
    {
        case 1:
                full_time_emp ft;
                // First read data
                ft.read_emp_data( );
                ft.calculate_salary( );
                // Now display data
                ft.display_emp_data( );
                break;
        case 2:
                part_time_emp pt;
                // First read data
                pt.read_emp_data( );
                pt.calculate_salary( );
                // Now display data
                pt.display_emp_data( );
                break;
    }
}
```

Output:

```
You have following choices
1-Read & display information of full time employee
2-Read & display information of part time employee
Enter your choice=1

Enter name of employee = Prajakta
Enter working rate for one day = 250
Enter no. of working days of employee=30

For the full time employee:=
Name=Prajakta
Daily rate =250
No. of working days =30
Salary =7500
```

- In the above program, both derived classes have same names for the member functions. You can also use different names.
- Also in the above program, you can use array of objects to store information about more than one employee.

5.2.5 Hybrid Inheritance [W-18]

- Sometimes, we may require to combine two or more types of inheritance to design a program; the result is known as Hybrid Inheritance. Fig. 5.11 illustrates this point.

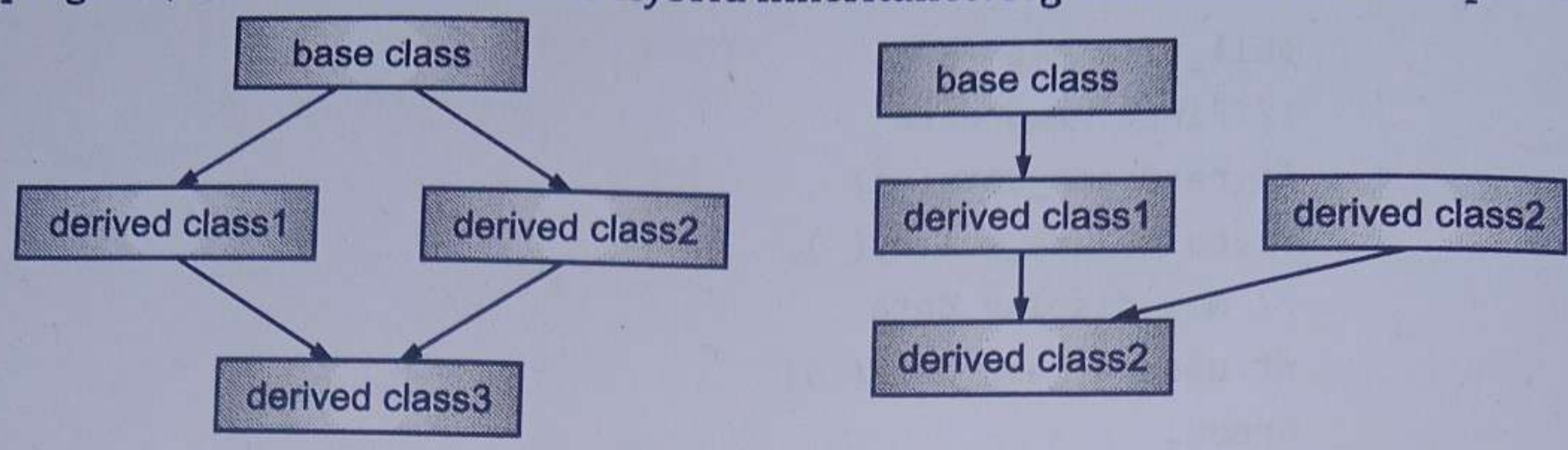


Fig. 5.11: Hybrid Inheritance

- **Example:** Consider an example where we have three classes' student, test and result. But some authority wants that some weightage should be given for sports to finalize the result. This is shown in Fig. 5.12.

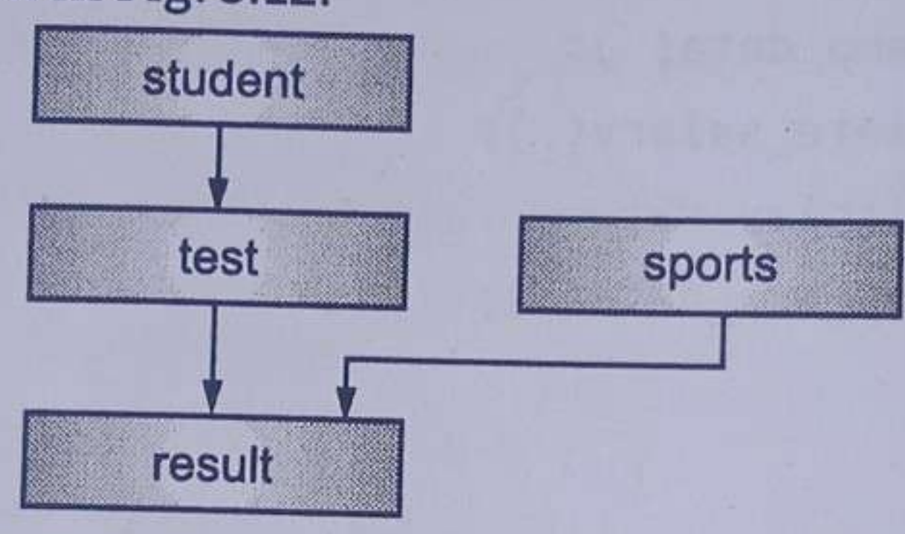


Fig. 5.12: Hybrid Inheritance (Multiple and Multilevel)

Program 5.8: Program for hybrid inheritance.

```
#include<iostream.h>
class student
{
   protected:
      int rollno;
   public:
      void get (int x)
      { rollno = x; }
      void display()
      { cout<<rollno; }
};
class test: public student
{
   protected:
      int mark1, mark2;
```

```
    public:
        void getmark(int a, int b)
        {
            mark1 = a;
            mark2 = b;
        }
        void dispmark()
        { cout<<mark1<<"  "<<mark2; }
};
class sport
{
    protected:
        int smark;
    public:
        void getsm(int s)
        { smark = s; }
        void disps()
        { cout<<smark<<"\n"; }
};
class result: public test, public sport
{
    private:
        int total;
    public:
        void displaytotal()
        {
            total = mark1 + mark2 + smark;
            display();
            dispmark();
            disps();
            cout<<"total = "<<total;
        }
};
void main()
{
    result r;
    r.get(10);
    r.getmark(50, 60);
    r.getsm(70);
    r.displaytotal();
}
```

Output:

```
10    50    60    70
total = 180
```

5.3 BASE CLASS AND DERIVED CLASS EXAMPLES

- In C++, a class that is inherited is referred to as a base class. The class that does the inheriting is called the derived class.
- A derived class can be defined by specifying its relationship with the base class. The pictorial representation of single inheritance is shown in Fig. 5.13.

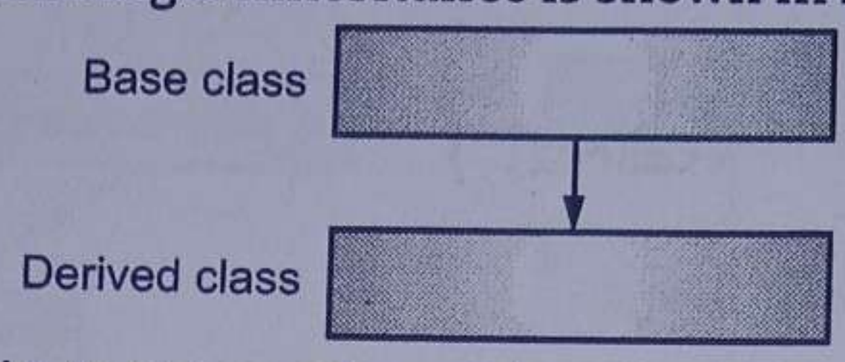


Fig. 5.13: Base Class and Derived Class

- Some example of inheritance is shown in Table 5.1.

Table 5.1: Examples of inheritance

Base Class	Derived Class
Shape	Circle Triangle Rectangle
Employee	Teacher Clerk
Media	Book Tape
Student	Undergraduate Postgraduate

- When a class inherits another, the member of the base class becomes members of the derived class.
- The **syntax** of derived class definition is as follows:

```
class <derived_class_name>: <access modifier> <base_class_name>
{
    // body of class
};
```

It contains:

- Keyword class.
- Derived class name which is user define name for new class.
- The : (colon) indicates that the derived_class_name is derived from the base_class_name.
- The access modifier is optional. If no access modifier is present, the access modifier is private by default. If written, then it must be public, private or protected.
- base_class_name is the existing class name.

- **Example:**

```
class base
{
    ..............
    ..............
};
class derived1: private base          //private derivation
{
    ..............
    ..............
};
class derived2: public base           //public derivation
{
    ..............
    ..............
};
class derived3: protected base        //protected derivation
{
    ..............
    ..............
};
class derived: base                   //by default private derivation
{
    ..............
    ..............
};
```

5.3.1 Public Derivation

- When the access modifier for a base class is **public**, all public members of the base class become public members of the derived class and all protected members of the base class becomes protected members of the derived class.
- The members of derived class can access members of the base class if the base class members are public or protected. They can not access private members.
- The Fig. 5.14 shows the base and derived class relationship.

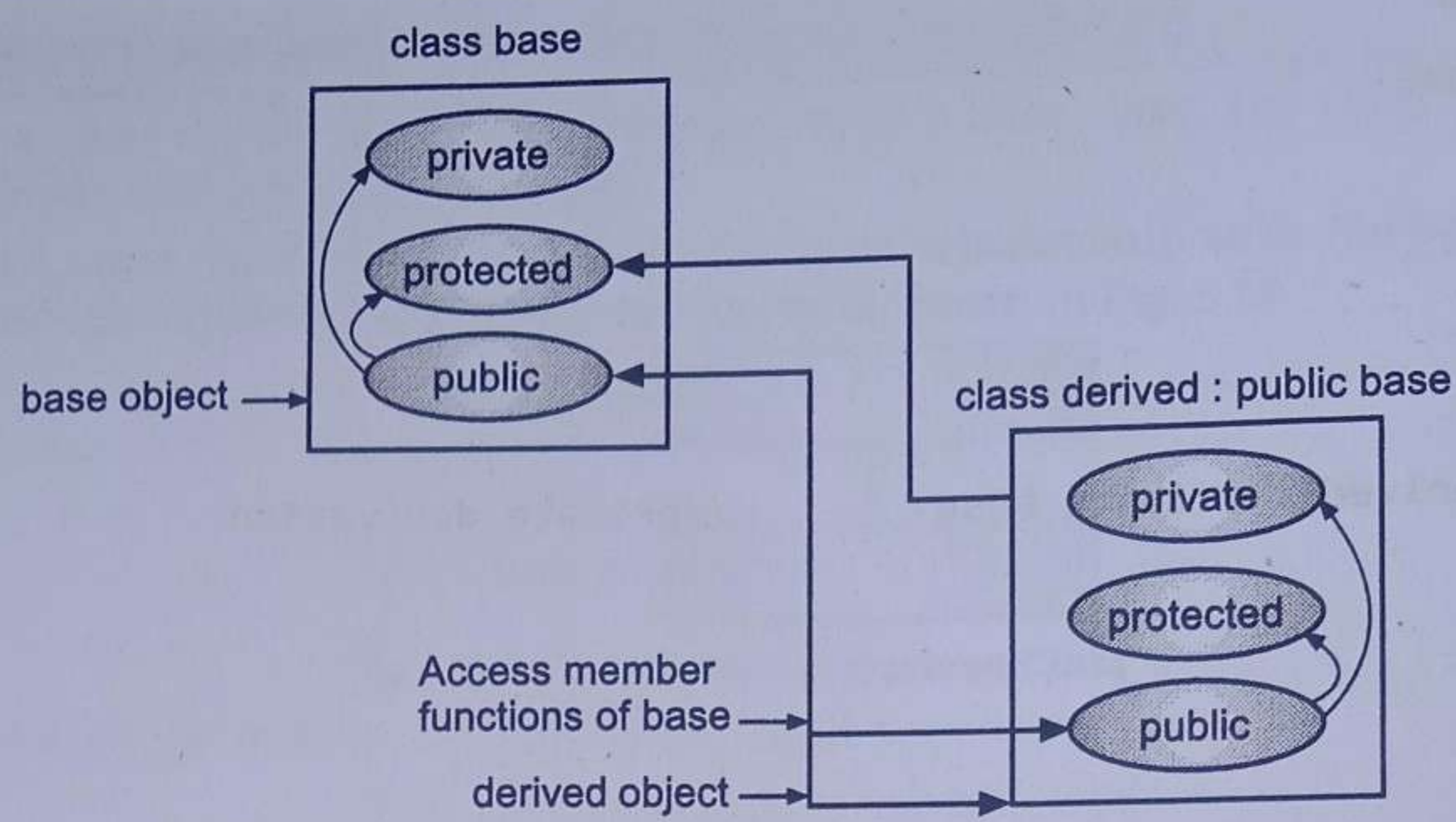


Fig. 5.14: Public Derivation

Program 5.9: Program to demonstrate public derivation.

```
#include<iostream>
using namespace std;
class base
{
    int i, j;
    public:
        void getij()
        {
            cout<<"Enter value of i and j";
            cin>>i;
            cin>>j;
        }
        void show()
        {
            cout<<"i="<<i<<"j="<<j<<"\n";
        }
};
class derived: public base
{
    int k;
    public:
        void getdata()
        {
            getij();
                cout<<"Enter value of k";
```

```
            cin>>k;
            cout<<"k="<<k;
        }
};
int main()
{
    base ob1;           //base object
    derived ob2;        //derived object
    ob1.getij();        //access members of base class
    ob1.show();         //display values of i and j
    ob2.getdata();      //access members of derived class
    ob2.show();         //access members of base class
}
```

Output:

```
Enter value of i and j 2 3
i=2 j=3
Enter value of i and j 5 6
Enter value of k 7
k=7 i=5 j=6
```

- In this program, the base class members are directly access through derived class, because the derived through derived class and derived class is publically derived from base class.
- In inheritance some of the base class data elements and member functions are inherited into the derived class which extends the capability of the existing classes and useful in incremental program development. If the function names of the base and derived class are same then use scope resolution operation (::) and show the scope of the function i.e. to which the class that function belongs.

Program 5.10: Write a program to read data members of a base class i.e. name, person id and the derived class members are height and weight. Display the data of person.

```
#include<iostream>
using namespace std;
class base
{
    private:
    char name[20];
    int person_id;
public:
    void getdata()
    {
        cout<<"Enter name and person_id";
        cin>>name>>person_id;
    }
```

```
    void display()
    {
        cout<<"name:"<<name<<"  "<<"person_id:"<<person_id;
    }
};
class derived: public base
{
    private:
        int height, weight;
    public:
        void getdata()
        {
            base:: getdata();
            cout<<"Enter height and weight";
            cin>>height>>weight;
        }
        void display()
        {
            base::display();
            cout<<"height:"<<height<<" "<<"weight:"<<weight;
        }
};
int main()
{
    derived obj;
    obj.getdata();
    obj.display();
}
```

Output:

```
Enter name and person_id prajakta 11
Enter height and weight 6 80
name: prajakta  person_id:11 height:6  weight:80
```

5.3.2 Private Derivation

- When the base class is inherited by using the private access modifier, all public and protected members of the base class become private members of the derived class.
- The Fig. 5.15 shows the private derivation of inheritance.

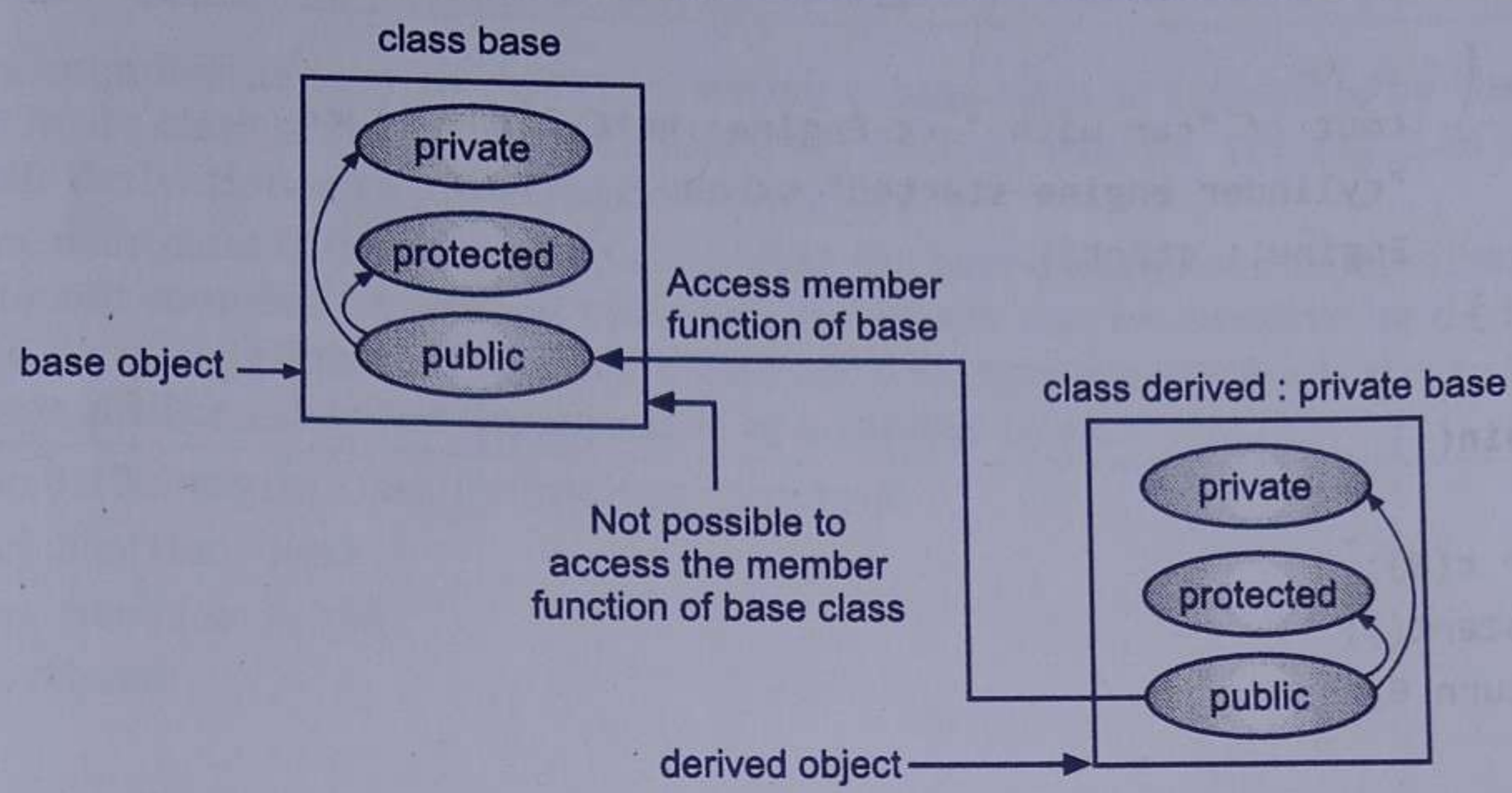


Fig. 5.15: Private Derivation

Program 5.11: Program for private derivation.

```
#include <iostream>
using namespace std;
class Engine
{
    public:
        Engine(int nc)
        {
            cylinder = nc;
        }
        void start()
        {
            cout << getCylinder() <<" cylinder engine started" << endl;
        }
        int getCylinder()
        {
            return cylinder;
        }
    private:
        int cylinder;
};
class Car: private Engine
{
    public:
        Car(int nc = 4): Engine(nc) { }
        void start()
```

```
        {
            cout << "car with " << Engine::getCylinder() <<
            "cylinder engine started" << endl;
            Engine:: start();
        }
};
int main( )
{
    Car c(8);
    c.start();
    return 0;
}
```

Output:

```
car with 8 cylinder engine started
8 cylinder engine started
```

- In this program, derived class object cannot access the functions of base class because they become private. But the public functions of base class access by public functions of derived class.

5.3.3 Protected Members

- The protected keyword provides greater flexibility in the inheritance mechanism. When a member of a class is declared as protected, that member is not accessible by other, non-member elements of program.
- Protected members are as good as private members within the class. Protected member differs from private members when a protected member is inherited. Using the access modifier protected in base class, we could make available the data member in the derived class and restrict its access in other classes or in main().

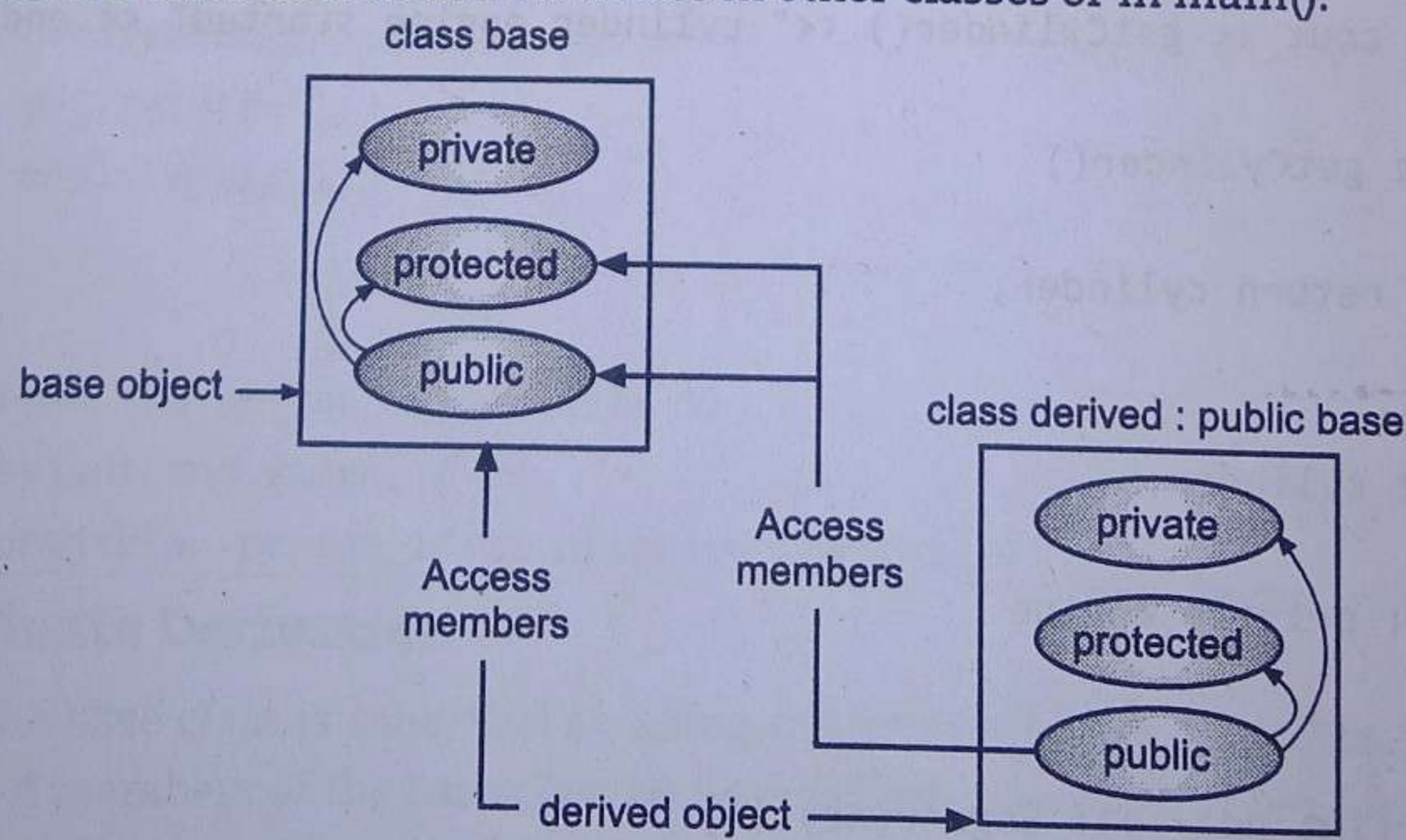


Fig. 5.16: Protected members of base class can be access in derived class

- Data member declare as protected within a base class is accessible by the member within its class and any class immediately derived from it. The Fig. 5.16 shows the public derivation with protected data members.
- If the base class is inherited as public, then the base class protected member becomes protected member of derived class and therefore, can be accessed by derived class. Thus, by using **protected**, we can create class members that are private to their class but can still be inherited and accessed by a derived class.

Program 5.12: Program using protected members.

```
#include<iostream>
using namespace std;
class base
{
   protected:
      int i, j;    // private to base, but accessible by derived class
   public:
      void getij()
      {
         cout<<"Enter value of i and j";
         cin>>i>>j;
      }
      void show()
      {
         cout<<"i="<<i<<" "<<"j="<<j<<"\n";
      }
};
class derived1: public base
{
   int k;
   public:
      void getdata()
      {
         getij();
         k=i*j;
      }
      void showall()
      {
         cout<<"k="<<k;
      }
}; //end of class derived1
```

```
class derived2: public derived1
{
    int x;
    public:
        void get()
        {
            getij();
            x=i*j;
        }  // can access i and j since public inheritance
        void showx()
        {
            cout<<"x="<<x;

        }
}; //end of class derived2
int main()
{
    derived1 ob1;
    derived2 ob2;
    ob1.getdata();
    ob1.showall();
    ob2.get();
    ob2.showx();
}
```

Output:

```
Enter value of i and j 10 20
k=200
Enter value of i and j 15 10
x=150
```

- In the above program, if derived1 class inherits base class privately, then protected member of base becomes private members of derived1 class which cannot inherited in derived2 class. So, when a protected member is inherited in public mode, it becomes protected in the derived class and therefore it is accessible by the member function of the derived class. It is also inherited further.
- A protected member inherited in the private mode becomes private in the derived class and cannot be inherited further.

5.3.4 Protected Derivation

- C++ allows us to use protected as access modifier. When access modifier is protected, all public and protected members of the base class become protected members of the derived class.

Program 5.13: Program for protected derivation.

```
#include<iostream>
using namespace std;
class base
{
    protected:
        int i, j;   //private to base but accessible in derived
    public:
        void getij()
        {
            cin>>i>>j;
        }
        void show()
        {
            cout<<i<<" "<<j<<"\n";
        }
};
class derived: protected base
{
    int k;
    public:
        void getdata()
        {
            getij();
            k = i*j;   // derived may access base protected data members
        }
        void showall()
        {
            cout<<k<<"  ";
            show();

        }
};
int main()
{
    derived ob;
    //ob.getij();    //illegal, getij() is protected member of derived
    ob.getdata();    // correct as public member of derived
    ob.showall();    // correct as public member of derived
    //ob.show();     // illegal
}
```

Output:

```
2   3
6   2   3
```

- In this program, getij( ) and show() are public members of base, but they become protected members of derived. Using derived object is not accessible.

Summary about Access Specifiers:

Table 5.2: Access in a Derived Class

Base class members	Derived Class		
	Public derivation	Private derivation	Protected derivation
Private	not accessible	not accessible	not accessible
Protected	protected	private	protected
Public	public	private	protected

5.4 VIRTUAL BASE CLASS

- Consider a situation where multilevel, multiple and hierarchical all the three kinds of inheritance are involved. This is illustrate in Fig. 5.17.

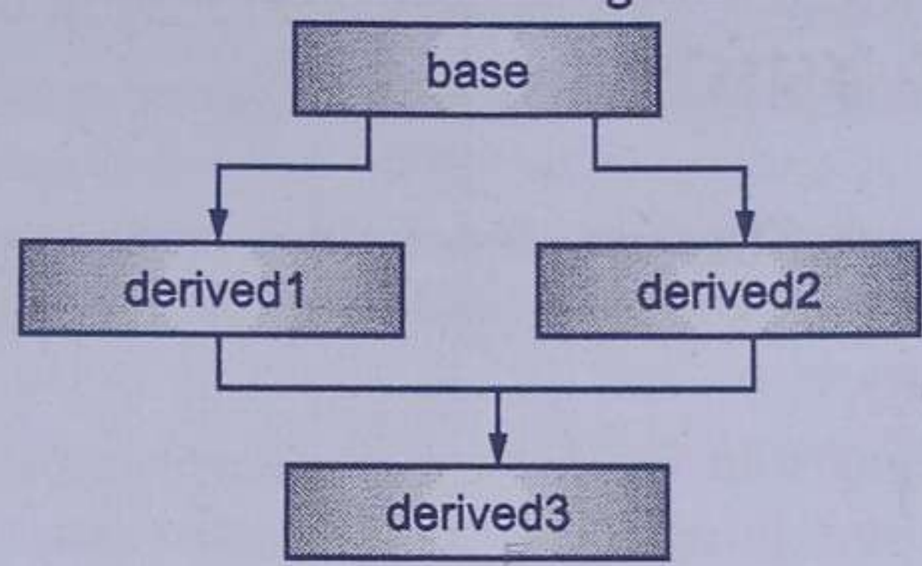


Fig. 5.17: Multipath Inheritance

- In the Fig. 5.17, the base class is inherited by both derived1 and derived2. derived3 directly inherits both derived1 and derived2. All the public and protected members of Base are inherited into derived3 twice through both derived1 and derived2. Therefore, derived3 would have duplicate sets of members inherited. This causes ambiguity when a member of Base is used by derived3.
- To resolve this ambiguity, C++ includes a mechanism by which only one copy of Base will be included in derived3. This feature is called a virtual base class. When a class is made virtual, C++ take necessary care to inherit only one copy of that class. The keyword virtual precedes the base class access specifier when it is inherited by a derived class.
- The situation in Fig. 5.17 can be declared as:

```
class base
{ ........... };
class derived1: virtual public Base
{ ........... };
class derived2: virtual public Base
{ ........... };
class derived3: public derived1, public derived2
{ ........... };
```

Program 5.14: Program using virtual base class.

```
#include<iostream.h>
//using namespace std;
class base
{
   public:
   int x;
};
class derived1: virtual public base
{
   public:
   int y;
};
class derived2: virtual public base
{
   public:
   int z;
};
/* derived3 inherits derived1 and derived2. However, only one copy of base
is present */
class derived3: public derived1, public derived2
{
   public:
   int mult()
   { return x * y * z; }
};
void main()
{
   derived3 d;
   d.x = 10;
   d.y = 20;
   d.z = 30;
   cout<<"product is"<<d.mult();
}
```

Output:

```
product is 6000
```

Note:

- If derived1 and derived2 classes had not inherited base as virtual, then statement d.x = 10; becomes ambiguous and a program would have resulted as compile-time error.

- If virtual base classes are used when an object inherits the base more than once, only one base class is present in the object.
- If normal base classes are used when an object inherits the base more than once, the multiple copies will be found; and it is compile time error.

Program 5.15: To calculate the total mark of a student using the concept of virtual base class.

```
#include <iostream>
using namespace std;
class student
{
  int rno;
  public:
  void getnumber()
  {
      cout<<"Enter Roll No:";
      cin>>rno;
   }
   void putnumber()
   {
   cout<<"\n\n\tRoll No:"<<rno<<"\n";
   }
};
class test:virtual public student
{
  public:
   int part1,part2;
   void getmarks()
   {
       cout<<"Enter Marks\n";
      cout<<"Part1:";
      cin>>part1;
      cout<<"Part2:";
       cin>>part2;
    }
    void putmarks()
    {
       cout<<"\tMarks Obtained\n";
       cout<<"\n\tPart1:"<<part1;
       cout<<"\n\tPart2:"<<part2;
    }
};
```

```
class sports:public virtual student
{
  public:
    int score;
    void getscore()
    {
       cout<<"Enter Sports Score:";
      cin>>score;
    }
    void putscore()
    {
      cout<<"\n\tSports Score is:"<<score;
    }
};
class result:public test,public sports
{
   int total;
   public:
   void display()
   {
      total=part1+part2+score;
      putnumber();
      putmarks();
      putscore();
      cout<<"\n\tTotal Score:"<<total;
   }
};
int main()
{
   result obj;
   obj.getnumber();
   obj.getmarks();
   obj.getscore();
   obj.display();
}
```

Output:

```
Enter Roll No: 151
Enter Marks

Part1: 90
Part2: 80
Enter Sports Score: 80
```

```
Roll No: 151
Marks Obtained
Part1: 90
Part2: 80
Sports Score is: 80
Total Score is: 250
```

5.5 ABSTRACT CLASS

- A class that contains at least one pure virtual function is said to be abstract. A pure virtual function is a member function i.e., declared in an abstract class, but defined in a derived class.
- An abstract class does not create any object and it contains one or more functions for which there is no definition (i.e., a pure virtual function).
- We can create pointers and references to an abstract class. This allows abstract classes to support runtime polymorphism, which depends upon base class pointers and references to select the proper virtual function.
- An abstract class is used only to derive other classes. For example: Shape is abstract base class which derives square, circle and rectangle.

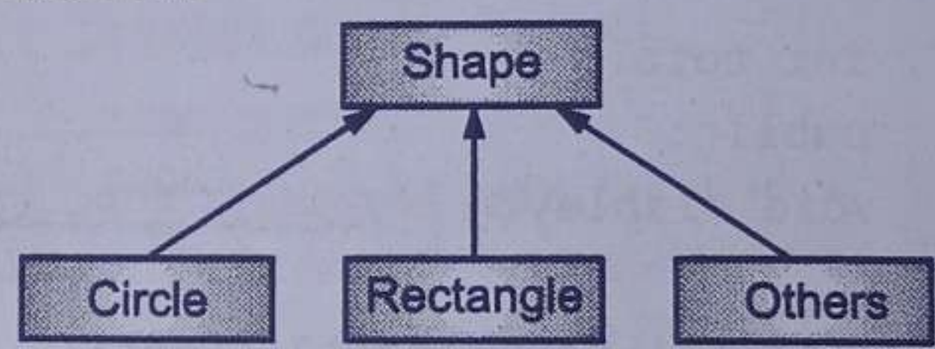


Fig. 5.18

Program 5.16: Following program shows use of abstract classes.

```
#include<iostream>
using namespace std;
// Abstract class
class Shape
{
    protected:
        float l;
    public:
        void getData()
        {
            cin >> l;
        }
        // virtual Function
        virtual float calculateArea() = 0;
};
class Square: public Shape
{
```

```
    public:
        float calculateArea()
        {   return l*l;  }
};
class Circle: public Shape
{
    public:
        float calculateArea()
        { return 3.14*l*l; }
};
int main()
{
    Square s;
    Circle c;
    cout << "Enter length to calculate the area of a square: ";
    s.getData();
    cout<<"Area of square: " << s.calculateArea();
    cout<<"\nEnter radius to calculate the area of a circle: ";
    c.getData();
    cout << "Area of circle: " << c.calculateArea();
    return 0;
}
```

Output:

```
Enter length to calculate the area of a square: 4
Area of square: 16
Enter radius to calculate the area of a circle: 5
Area of circle: 78.5
```

5.6 POLYMORPHISM [S-19]

- Polymorphism means "one name, many forms". Polymorphism is the process of defining a number of objects at different classes into group and call the method to carry out the operation of the objects using different function calls.
- There are two types of polymorphism:

1. Compile time polymorphism: [S-18]

- This is also called as **early** or **static** binding. Selection of an appropriate function for a particular call at the compile time itself. For example, function overloading and operator overloading. Function overloading is also referred to as functional polymorphism. The same function can perform a wide variety of tasks.
- The same function can handle different data types. When many functions with the same name but different argument lists are defined, then the function to be invoked corresponding to a function call is known during compile time.

- When the source code is compiled, the functions to be invoked are bound to the compiler during compile time, as to invoke which function depending upon the type and number of arguments. Such a phenomenon is referred to early binding, static linking or compile time polymorphism.

2. **Run-time polymorphism:**

- This is also called as **dynamic binding** or **late binding**. Sometimes, a situation occurs where function name and prototype is same in both the base class and in the derived class. Compiler does not know what to do, which function to call. In this class appropriate member function is selected to run time. For example, virtual function.

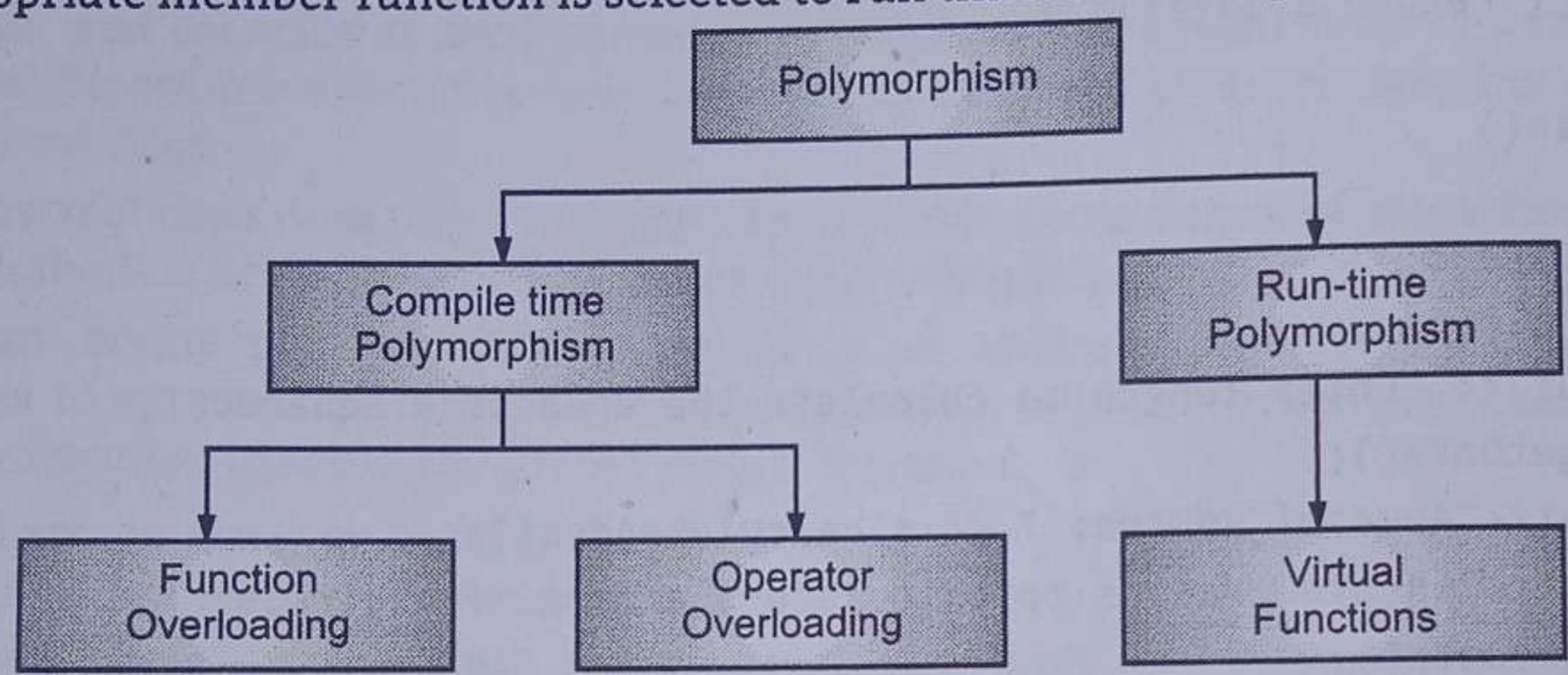


Fig. 5.19: Types of Polymorphism

5.7 VIRTUAL FUNCTIONS AND PURE VIRTUAL FUNCTIONS [W-18, W-19]

5.7.1 Virtual Functions [W-18, S-19]

- Virtual function is a member function that is declared within a base class and redefined by a derived class. A virtual function is declared by preceding the function declaration in the base class with the keyword **virtual.**

  Example: `virtual void show(); //show function is virtual.`

- When virtual functions access normally, it behaves just like any other type of class member function. However, virtual function is used to support runtime polymorphism when it is accessed via a pointer. A base class pointer can be used to point to an object of any class derived from that base. When different objects are pointed to different versions of the virtual functions are executed. Following program shows the usage of it.

Program 5.17: Program for virtual function.

```
#include<iostream.h>
class base
{
    public:
        virtual void display()
        { cout<<"This is base virtual function \n"; }
};
```

```
class derived1: public base
{
    public:
        void display()
        { cout<<"This is derived1 virtual function \n"; }
};
class derived2: public base
{
    public:
        void display()
        { cout<<"This is derived2 virtual function \n"; }
};
void main()
{
    base * p, b;      // p is base class pointer and b is object of
    derived1 d1;      base
    derived2 d2;
    p = &b;           // point to base
    p → display();    // access base function
    p → & d1;         // point to derived1
    p → display();    // access derived1 function
    p = &d2;          // point to derived2
    p → display();    // access derived2 function
}
```

Output:

```
This is base virtual function
This is derived1 virtual function
This is derived2 virtual function
```

Rules for Virtual Functions:

- The virtual function must be member of same class.
- Virtual functions are accessed by using object pointers.
- They cannot be static members i.e. must be non-static members.
- The prototype of the base class version of a virtual function and all the derived class versions must be identical.
- We can have virtual destructors but do not virtual constructors.
- We cannot use a pointer to a derived class to access object of the base type i.e. reverse is not true.
- If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class.

- A virtual function can be a friend of another class.
- Because of the restrictions and differences between function overloading and virtual function redefinition, the term overriding is used to describe virtual functions redefinition by a derived class.

5.7.2 Pure Virtual Functions [S-18]

- Most of the times, the virtual function inside the base class is rarely used for performing any task. It only serves as a placeholder. Such a functions are called do-noting functions; which are pure virtual functions.
- A pure virtual function is a virtual function that has no definition within the base class.
- **Syntax:**

```
virtual <return_type> <function_name> <arg_list> = 0;
```

- When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, compile time error will occur. A class containing such a pure virtual function is called an **Abstract Class.**
- We can define pure virtual function as "The functions which are only declared but not defined in the base class are called as pure virtual function".

Properties of Pure Virtual Functions:

- A pure virtual function has no implementation in the base class, hence a class with pure virtual function cannot be instantiated.
- A pure virtual member function can be invoked by its derived class.
- It is just placeholder for derived class. The derived class is supposed to fill this empty function.

Program 5.18: Program for a pure virtual function.

```
#include <iostream>
using namespace std;
class number
{
    protected:
        int val;
    public:
        void set_val(int i)
        {
            val = i ;

        }
        virtual void show( ) = 0;     // pure virtual function
};
```

```
class hextype: public number
{
    public:
        void show( )
        {
        cout<<hex<<val<<"\n";
        }
};
class dectype: public number
{
    public:
        void show( )
        {
            cout<<val<<"\n";
        }
};
class octtype: public number
{
    public:
        void show()
        {
            cout<<oct<<val<<"\n";
        }
};
int main ()
{
    dectype d;
    hextype h;
    octtype o;
    d.set_val(20);
    d.show();    // displays 20 - decimal
    h.set_val(20);
    h.show();    // displays 14 - hexadecimal
    o.set_val(20);
    o.show();    // displays 24 - octal
    return 0;
}
```

Output:

```
20
14
24
```

- In the above program, as shown the base class, **number** contains an integer called **val**, the function **setval( )** and the pure virtual function **show( )**. The derived classes **hextype**, **dectype** and **octtype** inherit **number** and redefine **show( )** so that it outputs the value of **val** in each respective number base (i.e. hexadecimal, decimal or octal).

5.8 OVERRIDING

- If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism.
- It is often useful for a derived class to define its own version of a member function inherited from its base class. This may be done to specialize the member function to the needs of the derived class. When this happens, the base class member function is said to be overridden by the derived class.
- Because of the restrictions and differences between function overloading and virtual function redefinition, the term overriding is used to describe virtual functions re-definition (overridden) by a derived class.
- In simple words, the re-definition of a virtual function in a derived class is usually called overriding.

Program 5.19: Program for overriding of member functions.

```
#include <iostream>
using namespace std;
class BaseClass
{
   public:
      void disp()
      {
         cout<<"Function of Parent Class";
      }
};
class DerivedClass: public BaseClass
{
   public:
   void disp()
   {
      cout<<"Function of Child Class";
   }
};
int main()
{
   DerivedClass obj = DerivedClass();
   obj.disp();
   return 0;
}
```

Output:

```
Function of Child Class
```

Summary

- Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.
- The technique by which features and properties of an old class is transferred into a new class is called inheritance or derivation. The old class is referred to as the base class or super class and new one is called the derived class or subclass.
- A derived class with only one base class is called single inheritance and one with several base classes is called multiple inheritance. A class may be inherited by more than one class is known as hierarchical inheritance.
- When a class is derived from another one of more base classes, this process is termed hybrid inheritance.
- The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance.
- When classes are declared virtual, the compiler takes essential caution to avoid duplication of member variables. Thus, we make a class virtual if it is a base class that has been used by more than one derived class as their base class.
- When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.
- When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class.
- A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.
- When deriving from a private base class, public and protected members of the base class become private members of the derived class.
- A class containing a pure virtual function is called an abstract class. It is called abstract because we cannot define objects of a class containing a pure virtual function. It exists only for the purpose of defining classes that are derived from it.
- Polymorphism means "one name, many forms". Polymorphism is the process of defining a number of objects at different classes into group and call the method to carry out the operation of the objects using different function calls.
- There are two types of polymorphism:
  1. **Compile time polymorphism:** This is also called as early or static binding selection of an appropriate function for a particular call at the compile time itself.
  2. **Run-time polymorphism:** This is also called as dynamic binding or late binding. Sometimes, a situation occurs where function name and prototype is same in both the base class and in the derived class.
- A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
- A pure virtual function is a virtual function in C++ for which we need not to write any function definition and only we have to declare it. It is declared by assigning 0 in the declaration. An abstract class is a class in C++ which have at least one pure virtual function.

Check Your Understanding

1. The process of creating new classes from an existing class is called as _____.
   (a) polymorphism (b) polyinheritance
   (c) inheritance (d) none
2. If a base class is privately inherited by a derived class become _____ member to derived class.
   (a) public (b) private
   (c) visible (d) none
3. A class which is used only to derive other classes is known as _____ class.
   (a) abstract (b) virtual
   (c) derived (d) none.
4. Hybrid inheritance is _____.
   (a) multiple inheritance (b) multilevel inheritance
   (c) multipath inheritance (d) both (a) & (b)
5. A class is inherited known as _____.
   (a) base class (b) hybrid
   (c) derived class (d) both (a) & (b)
6. Following _____ operator is used for resolve ambiguity.
   (a) + (b) ::
   (c) – (d) none of these
7. Inheritance helps in making a general class into a more specific class.
   (a) True (b) False
8. Inheritance facilitates the creation of class libraries.
   (a) True (b) False
9. Defining a derived class requires some changes in the base class.
   (a) True (b) False
10. Polymorphism means _____.
   (a) Many forms (b) only one form
   (c) hiding data (d) none of them

Answers

1. (c)	2. (b)	3. (a)	4. (d)	5. (a)	6. (b)	7. (b)	8. (a)	9. (b)	10. (a)

Practice Questions

Q.I Answer the following questions in short:

1. What is main advantage of Inheritance?
2. List out different types of inheritance.
3. What is the difference between the Base and Derived Classes?
4. What is a Virtual Base Class?
5. What is Virtual Function?

6. What is pure virtual function?
7. What is overriding?

Q.II Answer the following questions:

1. Explain different types of Inheritance with example.
2. What are advantages and disadvantages of declaring Inheritance?
3. What is the scope of the accessibility of variables in the protected section and in the private section of a class?
4. When can Multiple Inheritance lead to ambiguity?
5. How is a Single Inheritance different from Multiple Inheritance?
6. Write short note on:
   (i) Multiple Inheritance
   (ii) Base Class and Derived Class
7. What is Pure Virtual Function? Explain with suitable example.
8. What is Polymorphism? What is the difference between Compile Time and Runtime Polymorphism
9. Write short note on: (a) Virtual Function, (b) Pure Virtual Function.

Q.III Define terms:

1. Inheritance
2. Single level inheritance
3. Multiple level inheritance
4. Hierarchical inheritance
5. Hybrid inheritance
6. Polymorphism

Previous Exam Questions

Summer 2018

1. A class is inherited known as _____. **[1 M]**
   (i) Base class
   (ii) Hybrid
   (iii) Derived
   (iv) Both (i) & (ii)

Ans. (i) Base class **[1 M]**

2. What is compile time polymorphism? **[4 M]**

Ans. Refer to Section 5.6.

3. Explain pure virtual function with example.

Ans. Refer to Section 5.7.2.

4. What is inheritance? Explain single inheritance with example. [4 M]

Ans. Refer to Section 5.2.1.

5. Explain all the forms of inheritance. [5 M]

Ans. Refer to Section 5.2.

Winter 2018

1. What is the multiple inheritance? [1 M]
   (a) when parent class has two or more child classes
   (b) when base class has two or more derived classes
   (c) when a child class has two or more parent class
   (d) where two classes inherit from each other

Ans. Refer to Section 5.2.3.

2. Explain hybrid inheritance with the help of an example. [4 M]

Ans. Refer to Section 5.2.5.

3. Give difference between virtual and pure virtual functions. [3 M]

Ans. Refer to Section 5.7.

4. What is an inheritance? What ambiguity can arise in multiple inheritance? How is it solved? Give example. [5 M]

Ans. Refer to Section 5.2.3.

5. What is the purpose of virtual function? State the rules for virtual function. [4 M]

Ans. Refer to Section 5.7.1.

Summer 2019

1. In public inheritance the public and protected data members of the base class remain _____ in the derived class.
   (a) public
   (b) private
   (c) public and protected resp.
   (d) protected

Ans. Refer to Section 5.3.

2. What are the types of polymorphism? [1 M]

Ans. Refer to Section 5.6.

3. Give difference between virtual and pure virtual function. [3 M]

Ans. Refer to Section 5.7.

4. What is virtual function? Explain with example. [5 M]

Ans. Refer to Section 5.7.1.

5. Define Inheritance. Give advantages of it. [3 M]

Ans. Refer to Section 5.1.

6... Files and Exception Handling

Objectives...

- To learn Stream Classes for File operations.
- To study different File Operations.
- To understand File updating with Random Access.
- To learn Exception Handling.

6.1 INTRODUCTION

- It is difficult to handle large amount of data only with use of keyboard and screen. So, we need to store data on secondary storage device as data structures called files.
- A computer system stores programs and data in secondary storage in the form of files. A file is a collection of related information.
- A file is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user. A file is named and is referred to by its name.
- To define a file properly, it is necessary to consider the operations which can be performed on files. The Operating System (OS) provides most of the essential file manipulation services such as create, open, write, read, rewind, close and delete.
- A program typically involves data communication between the console and the program or between the files and program or even both. The program must at least perform data exchange between processor and main memory.
- Computer programs are written to store the data (a write operation on file) and to read the data (a read operation on file).
- A computer program involves following two types of data communication:
  - Data transfer between the keyboard/monitor (console) and the program.
  - Data transfer between the program and the disk file.

- Fig. 6.1 shows the concept of data communication.

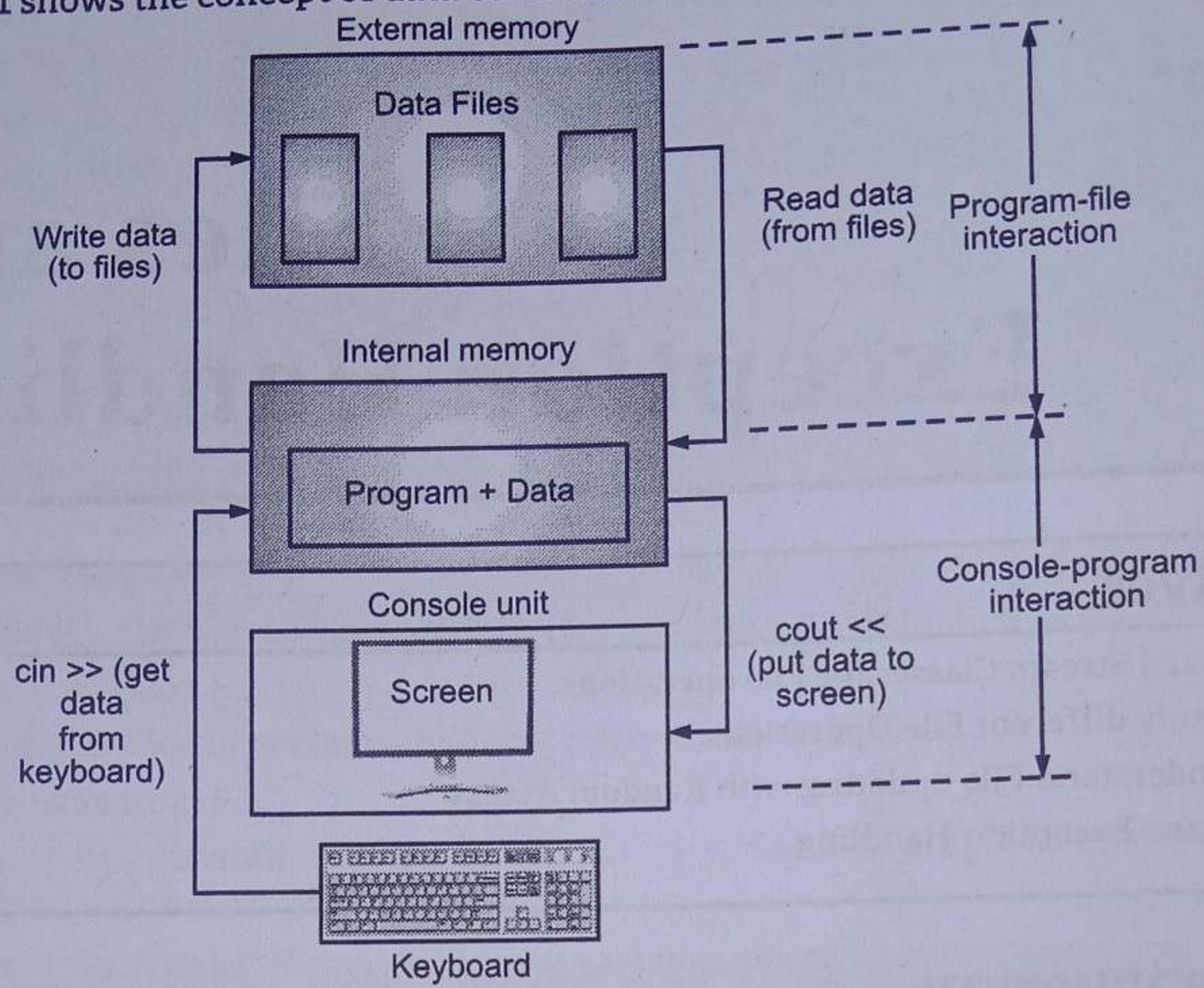


Fig. 6.1: Console Program and File Interaction

- C++ uses the concept of stream and stream classes to implement its I/O operations with the console and disk files.
- Files are either text files or binary files. A text files can be created using sophisticated word processor. Text files can be easily read or printed. Binary files are data files and program files.
- Binary files can not be easily read or write but size of binary file is smaller than that of text file. The speed at which programs can read data from binary file is very high as compare to text file.
- In this chapter, we will discuss various methods available for storing and retrieving the data from files. Using files we can transfer data from program to file on secondary storage or vice versa.

6.2 CLASSES FOR FILE STREAM OPERATIONS

- The file handling techniques of C++ support file manipulation in the form of stream objects. The stream object `cin` and `cout` are used extensively to deal with the standard input and output devices.
- The C++ Input/Output system support a hierarchy of classes that are used to manipulate both the console and disk files, called stream classes.

- The Input/Output system of C++ has a group of classes that defines all the file handling functions. These classes are:
  - `ifstream`: For handling input files.
  - `ofstream`: For handling output files.
  - `fstream`: For handling files on which both input and output can be performed.
- Input stream is declared as a class ifstream. It is derived from istream. Output stream is declared as a class ofstream. It is derived from ostream. Input/Output stream should belong to fstream class. It is derived from iostream.
- The hierarchy of C++ file stream classes is shown in Fig. 6.2.

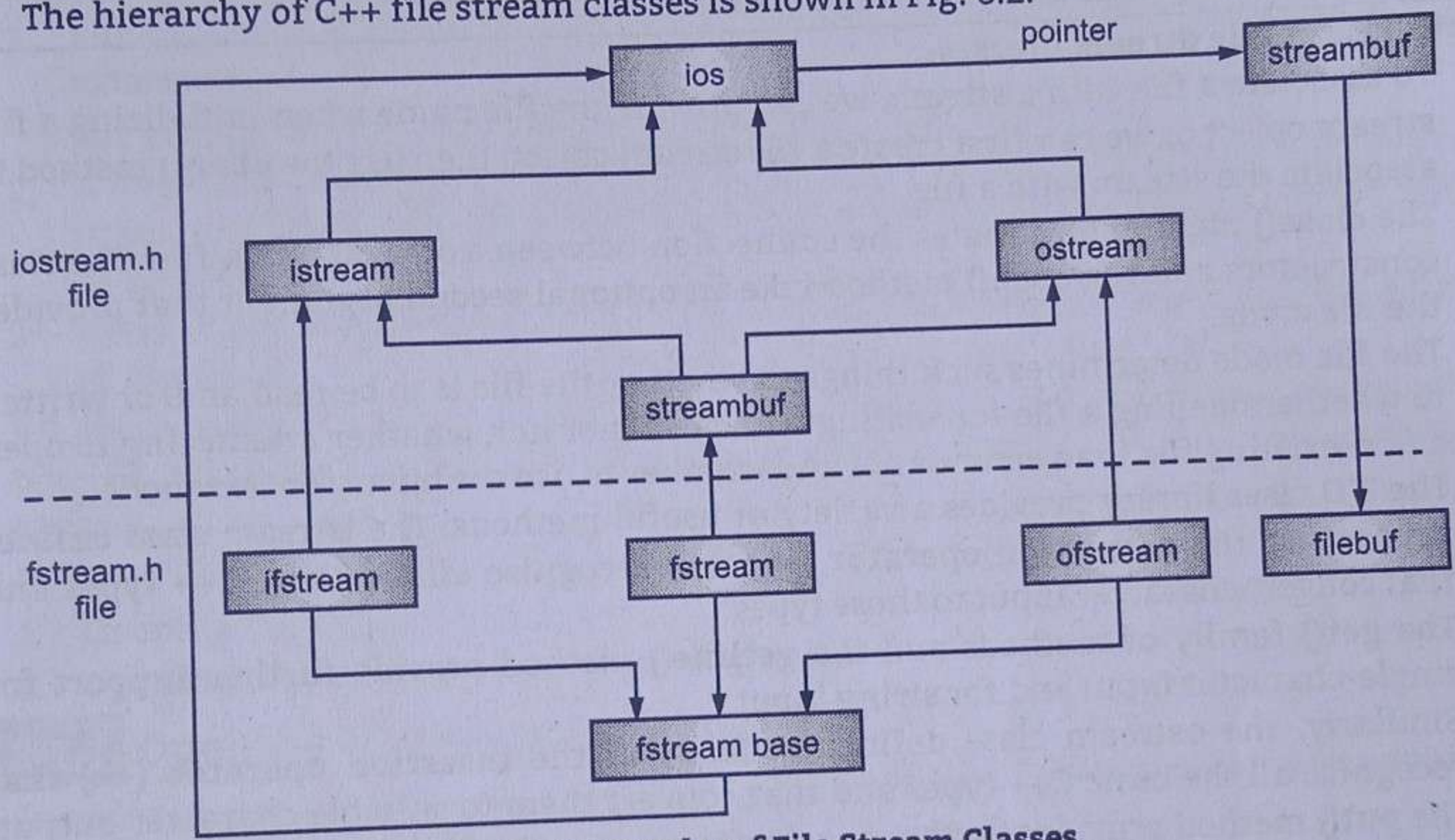


Fig. 6.2: Hierarchy of File Stream Classes

- The classes ifstream, ofstream, and fstream are designed exclusively to manage the disk files and their declaration exists in the header file fstream.h. To use these classes, include the following statement is used in the program, `#include<fstream.h>`.
- C++ views each file as a sequence of bytes. Each file ends either with an end of file marker or at a specific byte number. For file processing iostream.h and fstream.h must be included. fstream is used for read (ifstream) and write (ofstream).
- The functionalities of istream, ostream and iostream are inherited to ifstream, ofstream and fstream respectively. Details of file stream classes are given in the following table.

Table 6.1: Shows the details of file stream classes

Class	Contents
`filebuf`	Its purpose is to set the file buffers to read and write. Contains close() and open() as its members.
`fstreambase`	It is the base class of fstream, ifstream and ofstream classes. Contains close() and open() functions.

contd. ...

ifstream	It provides input operations. Contains open() with default input mode. Inherits the functions get(), getline(), read(), seekg() and tellg() from istream.
ofstream	It provides output operations. Contains open() with default output mode. Inherits put(), seekp(), tellp() and write() from ostream.
fstream	It allows simultaneous input and output operations. Contains open() with default input mode. Inherits all the functions from istream and ostream classes through iostream.

Methods for File Stream Classes:

- To associate a file with a stream, we can provide the file name when initializing a file stream object or we can first create a file stream object, then use the **open()** method to associate the stream with a file.
- The **close()** method terminates the connection between a stream and a file. The class constructors and the open() method take an optional second argument that provides the file mode.
- The file mode determines such things as whether the file is to be read and/or written to whether opening a file for writing truncates it or not, whether attempting to open a non-existing file is an error or not and whether to use the binary or text mode.
- The I/O class library provides a variety of useful methods. The istream class defines versions of the **extraction operator (>>)** that recognize all the basic C++ types and that convert character input to those types.
- The **get()** family of methods and the **getline()** method provide further support for single-character input and for string input.
- Similarly, the ostream class defines versions of the **insertion operator (<<)** that recognize all the basic C++ types and that convert them to suitable character output. The **put()** method provides further support for single-character output.
- A text file stores all information in character form. For example, numeric values are converted to character representations. The usual insertion and extraction operators, along with **get()** and **getline()**, support this mode.
- A binary file stores all information using the same binary representation the computer uses internally. Binary files store data, particularly floating point values, more accurately and compactly than text files, but they are less portable. The **read()** and **write()** methods support binary input and output.
- The **seekg()** and **seekp()** functions provide random access for files. These class methods let us to position a file pointer relative to the beginning of a file, relative to the end or relative to the current position. The **tellg()** and **tellp()** methods report the current file position.

6.3 FILE OPERATIONS

- There are various operations available on data files:
  1. Opening and closing files.
  2. Reading and writing files.
  3. Detecting End of file.
  4. Updating of file.

[S-18]

6.3.1 Opening a File

- Generally, if we are using files in the program, we must think about the name of file, the structure of file and opening mode of file.
- The file name is a string with at the most *eight* characters.
- It can have optional three characters for extension and both i.e. file name and extension are separated by period (•).
- To open a file:
  1. File stream is created and then linked to the file, which has two part: primary and extension.

     For example: x.cpp – Here x, is a filename and .cpp is extension.
  2. A filestream is defined by using classes ifstream, ofstream and fstream which are present in header file fstream.h. These classes are used depending on operations read or write.
  3. When a file is opened for write only, a new file is created if there is no file exists of that name.
- When we want to write certain text or information, the file should be created and opened, and even if we want to read the contents from the file, the file should be opened.
- In C++, a file can be opened in two ways:
  1. Opening file using constructor.
  2. Opening file using open( ) member function.

6.3.1.1 Opening File using Constructor

- In order to perform file operations, the file should be open. As we know, constructor is used to initialize an object while it is being created.
- Here, a constructor is used to initialize the file name which is used with the file stream object.
- The creation of file using file stream object involves the following steps:
  1. **Create a file stream object using stream classes as follows:**

```
ifstream ob1 ;          //input
ofstream ob2;           //output
fstream ob;             //input and output
```

  2. **File stream is identified by a file name:**

     For example, `ifstream ob1("stud.dat");`
- This statement declares **ob1** as an **ifstream** object and attaches it to the file **stud.dat** for reading (input) as shown in Fig. 6.3.

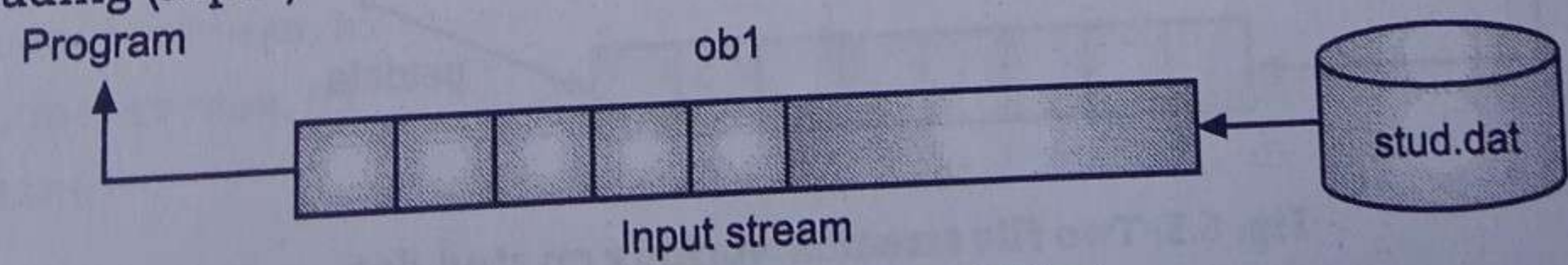


Fig. 6.3: Input Stream Object

- The statement `ofstream ob2("stud.dat");` opens a file in output mode i.e. in write mode. After opening a file, it is attached to output stream ob2 as shown in Fig. 6.4.

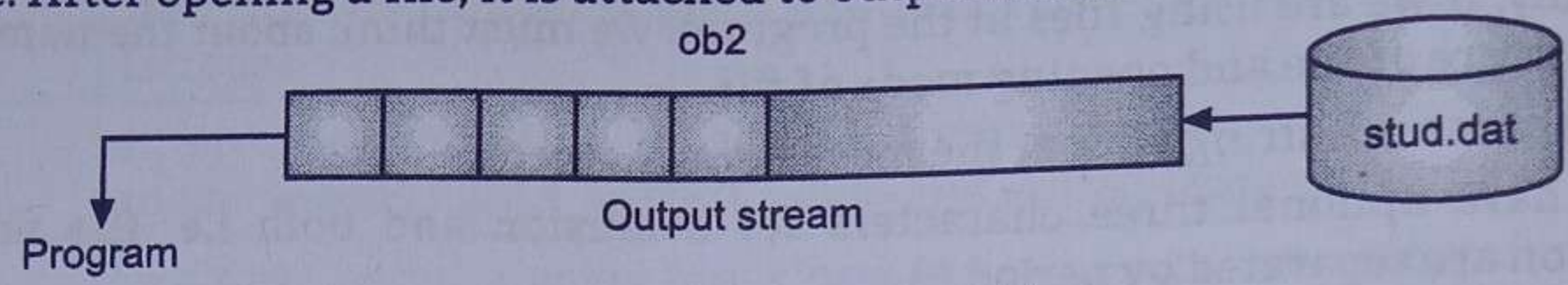


Fig. 6.4: Output Stream Object

Program 6.1: Program for open file in output mode.

```
#include<fstream.h>
#include<iostream.h>
int main()
{
    ofstream ob1("stud.dat");
    ifstream ob2 ("stud.dat");
    char str[80];
    //writing string to the file
    ob1<<"This is student database";
    //Read string from the file
    while (ob2)
    {
        ob2.getline(str, 80);        //read string using ob2
        cout<<str;                   //write on console
    }
    return 0;
}
```

Output:

```
This is student database
```

- Here ob1, ob2 are objects. Two different streams are working on the same file. This is shown in Fig. 6.5 where reading and writing are the operations performed on **stud.dat**.

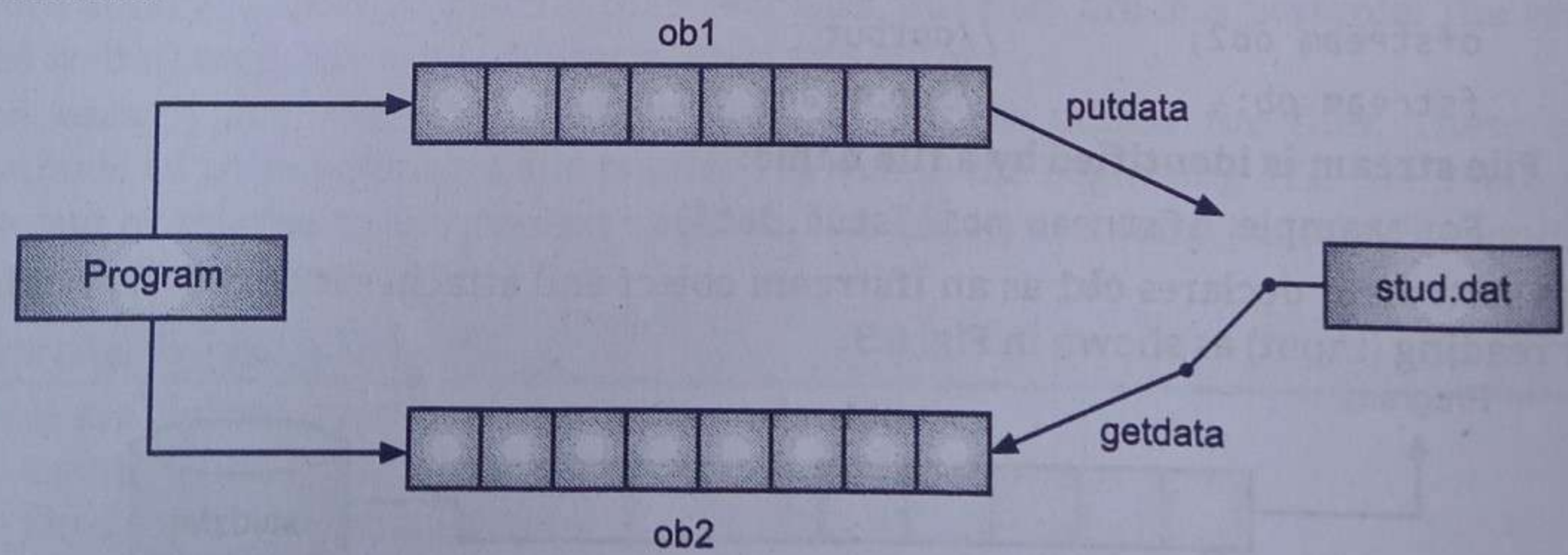


Fig. 6.5: Two file streams working on stud.dat

- When a stream object expires, the connection with a file is closed automatically i.e. when the program terminates, compiler automatically close a file. Here, to print the message into file we use the statement.

```
ob1 << "This is student database";
```

6.3.1.2 Open File Explicitly using Open( )

- The file is open explicitly using open() function instead of constructor.
- **The syntax for opening a file is:**

```
stream_class    stream_object;
stream_object.open ("filename");
```

For example:

```
ofstream ofile;                    //create stream for output
ofile.open("stud");                //connect stream to stud
--------------
--------------
ofile.close();                     //close the file stud
```

- The stream is connected to only one file at a time. Hence, using same stream object we cannot open many files simultaneously. If we want to open two files, think first file should be close before we open the second file. If we want to open many files same time then we can use different stream_objects to open files.

For example:

1.
```
ofstream ofile;
ofile.open ("Teacher");
..........
..........
ofile.close();
ofile.open("stud");
..........
..........
ofile.close();
```

2.
```
ofstream ofile, ofout;
ofile.open("Teacher");
ofout.open("stud");
..........
..........
```

Program 6.2: Program for opening two files with same stream and storing the contents in it.

```
#include<iostream.h>
#include<fstream.h>
int main()
{
    ofstream outfile;
```

```
    outfile.open("Empname");
    outfile<<"Ganesh\n";
    outfile<<"Mahesh\n";
    outfile<<"Harish\n";
    outfile<<"Dinesh\n";
    outfile.close();                    //closing the file
    outfile.open("ACC");                //opening the file
    outfile<<"001";
    outfile<<"022";
    outfile<<"333";
    outfile<<"444";
    outfile.close();
    //Reading from a file
    char line[20];
    ifstream infile;                    //create input stream
    infile.open("Empname");
    cout<<"Name of Employees:\n";
    while(infile)
    {
      infile>>line;                     //or infile.getline(line,20);
      cout<<line<<"\n";
    }
    infile.close();
    infile.open("ACC");
    cout<<"\n the account numbers:\n";
    while(infile)
    {
      infile>>line;
      cout<<line<<"\n";
    }
    infile.close();
  }
```

Output:

```
Name of Employees
Ganesh
Mahesh
Harish
Dinesh
The account numbers
001
022
333
444
```

More about open( ) file mode:

- C++ provides a mechanism of opening a file in different modes where second argument is explicitly passed.
- **syntax:**

```
stream_object.open("filename", openingmode);
```

- Here, the open( ) function takes two arguments, the first argument is file name and second argument is used for the file mode or file mode parameter, which are shown in Table. 6.2. We know that, ios::in & ios::out are default values for opening of ifstream and ofstream.
- The file modes are shown in Table. 6.2.

Table 6.2: File Open Modes

Parameter	Meaning
ios::ate	Go to end-of-file on opening.
ios::app	Append to end-of-file or all writes occurs at end of file.
ios::in	Open file for reading only.
ios::out	Open file for writing only.
ios::binary	Binary file (open file is binary mode).
ios::trunc	Delete contents of the file if it exists.
ios::nocreate	Open fails if file does not exist.
ios::noreplace	Open files if the file already exists.

- The various file modes in above table are explained below:
  1. **ios::ate:** It is used to sets a pointer to the end-of-file at opening. It allows to add or modify the existing data.
  2. **ios::app:** It is used to sets a pointer to the end-of-file similar to *ate*, but it add the data to the end of file.
  3. **ios::in:** This mode should be specified for input files.
  4. **ios::out:** This mode should be specified for output files.

     For the fstream class, mode should be explicitly specified.

     For example:
     ```
     fstream f;
     f.open("Emp", ios::in | ios::out);
     ```
  5. **ios::binary:** It is the most used mode for large applications and databases as compared to text file reading and writing.
  6. **ios::trunc:** It is a default mode if the file is opened for output.
  7. **ios::nocreate:** It is relevant with direct access files.
  8. **ios::noreplace**: This mode not allow us to open a file with the same name hence avoid destroying the previous file.
- We can separate more than one modes using bitwise OR or | symbol.

  For example, `out.open("Employee", ios::in | ios::app)`

Program 6.3: Program for ios::trunc file open mode.

```
#include<fstream.h>
#include<iomanip.h>
int main()
{
    float a = 122.25, b = 27, c = 33.5;
    ofstream fout;
    fout.open("num.data", ios::trunc);
    fout<<setw(6)<<a<<endl;
    fout<<setw(6)<<b<<endl;
    fout<<setw(6)<<c<<endl;
}
```

Output:

```
122.25
27.00
33.50
```

6.3.2 Closing a File

- To close a file we should use the input and output stream object name.
- For example,

```
ob1.close();
ob2.close();
```

- Here, the member function close() is used to close a file which has been open either for reading or writing or both purposes. The close() function is not contain any arguments. It is called explicitly as above. It also called automatically by the destructor function.
- Program 6.4, uses a single file for both writing and reading the data. First it takes data from the keyboard and writes it to the file. After writing is completed, the file is closed. The program again opens the same file, reads the information already written to it and displays the same on the screen.

Program 6.4: Program uses single file for both writing and reading data.

```
#include<iostream.h>
#include<fstream.h>
int main()
{
    ofstream ofile ("Employee");
    cout<<"Enter emp.name";
    char name[20];
    cin>>name;
    ofile <<name<<"\n";  //write to file employee
```

```
    cout<<"Enter salary of emp";
    double salary;
    cin>>salary;
    ofile<<salary<<"\n";
    ofile.close();
    ifstream ifile ("Employee");
    ifile>>name;
    ifile>>salary;
    cout<<"employee name:"<<name<<"\n";
    cout<<"employee salary:"<<salary<<"\n";
    ifile.close();
}
```

Output:

```
Enter emp name
Mahesh
Enter salary of emp
26500
employee name: Mahesh
employee salary:26500
Press any key to continue . . .
```

- When a file is opened for write only, a new file is created if there is no file exists of that name. If a file is exists already, then its contents are deleted.

6.4 RANDOM ACCESS FILES

- Random access means moving the file pointer directly to any location in the file instead of moving it sequentially.
- The random access technique is often used with the database files.
- Fig. 6.6 shows the classification of files.

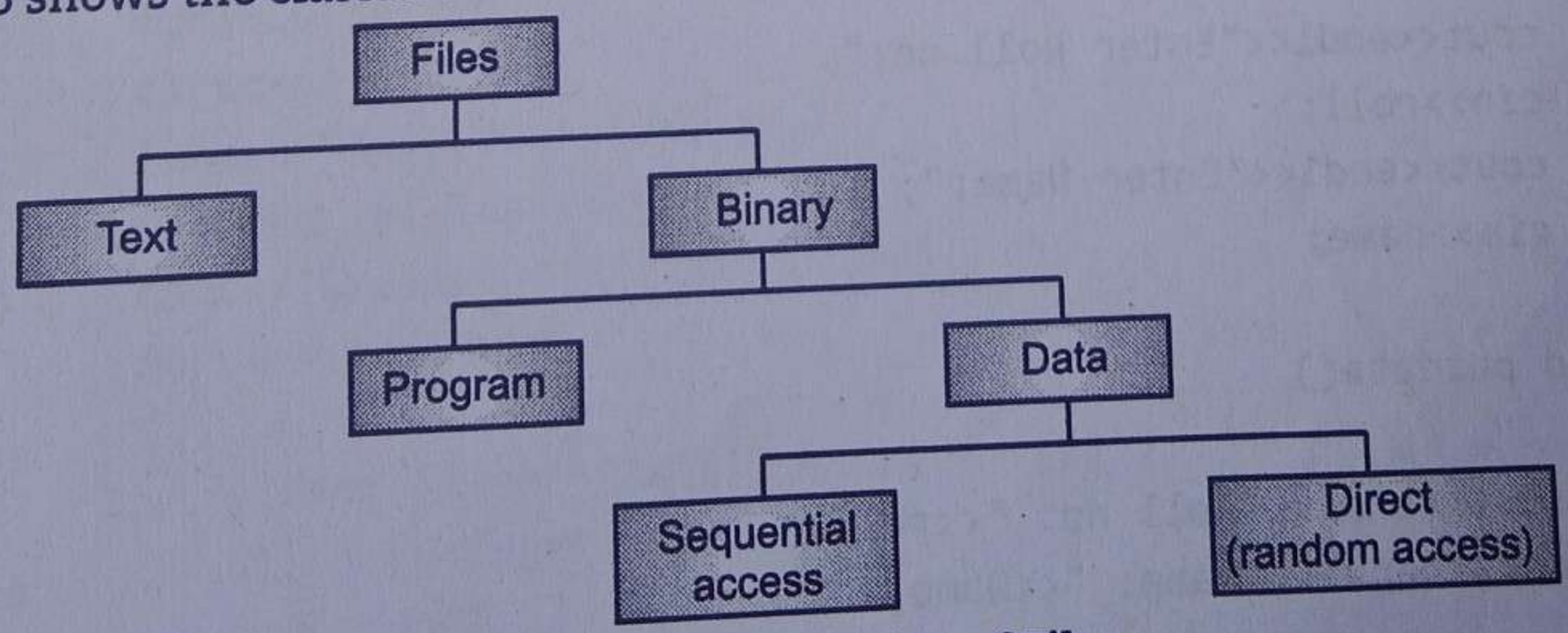


Fig. 6.6: Classification of Files

- In sequential access, we must open a file for reading and use file pointer to access records one by one sequentially.
- Binary data files can be accessed in direct or random manner. These files should be opened in binary mode.
- Random access performs the operations on file such as adding a new item, displaying the contents of a file, deleting an existing item, modifying an existing item, read and write.
- These requires that a file pointer to be move to an absolute position as specified by the parameter move.
- This can performed by using seekg( ), seekp( ), tellg( ), tellp( ), read( ) and write( ) functions. We use the following formulae in program.
- **Formula:**

1. To find size of object (*l*),

```
int l  = size of (object);
```

2. To find the location at which m^{th} object is stored,

```
int location = m * sizeof (object);
```

or `int location = m * l;`

3. To find total number of objects in file,

```
int n = filesize / sizeof (object);
```

Program 6.5: Program for random access approach.

```
//Basic File handling with add, modify, display operations
#include<iostream.h>
#include<fstream.h>
//using namespace std;
class student
{
   int roll;
   char name[20];
   public:
   void getdata()
   {
      cout<<endl<<"Enter Roll no:";
      cin>>roll;
      cout<<end1<<"Enter Name:";
      cin>>name;
   }
   void putdata()
   {
      cout<<end1<<"Roll no: "<<roll;
      cout<<end1<<"Name: "<<Name;
   };
```

```
int main()
{
    ifstream f;
    int n,i;
    char ch;
    student s[5],s1[5];
    clrscr();
    f.open("temp.txt",ios::out   ios::in);
    cout<<"writing data into file"
    for (i=0;i<2;i++)
    {
        s[i].getdata();
        f.write((char*)&s[i],sizeof(s[i]));
    }
    f.seekg(0);
    cout<<"reading from file & display on console"
    for(i=0;i<2;i++)
    {
        f.read((char*)&s[i],sizeof(s1[i]));
        s1[i].putdata();
        cout<<endl<<endl;
    }
    f.close();
    cout<<endl<<endl<<"Do you wish to update the file?(Y/N):";
    cin>>ch;
    cout<<"ADDING A RECORD"
    if(ch=='Y'||  ch=='y')
    {
        f.open("temp.txt",ios::app || ios::out);
        s[i].getdata();
        f.write((char*) &s[i],sizeof(s[i]));
        f.close();
        clrscr();
        f.open("temp.txt",ios::in);
        f.seekg (0, ios::beg);
        cout<<"Updated file is::"<<endl;
        for(i=0;i<3;i++)
        {
            f.read ((char*)&s1[i],sizeof(s1[i]));
            s1[i].putdata();
            cout<<endl;
        }
```

```
        cout<<"finding filesize & no. of objects"
        int l = f.tellg();
        int m = l/size of (s1[pi]);
        cout<<"no.of objects="<<m<<endl;
        cout<<"file size = "<<l<<"bytes"<<endl;
        f.close();
    }
    else
        f.close():
        return 0;
    }
```

Output:

```
Writing data into file
    200Riya  //file contents
    201Rucha
reading from file and display on console
    200Riya
    201Rucha
Do you wish to update the file? (Y/N): Y
ADDING A RECORD
    202Deepa
updated file is:
    200Riya
    201Rucha
    202Deepa
finding file size & no. of objects
no. of objects = 3
file size = 9 bytes
```

6.5 INTRODUCTION TO EXCEPTION HANDLING MECHANISM

- While writing and compiling C++ programs various errors may occur. These errors are:
  1. **Syntax errors** occur due to violation of syntax rules in C++. Programs having syntax errors do not execute.
  2. **Logical errors** occur due to wrong use of logics, i.e. formulae. Programs having logical errors can execute and compile, but the result obtained is erroneous.
  3. **Run-time errors** occur when the operations can not be performed during execution. Programs having run-time errors can compile but cannot produce meaningful results.
- For these errors C++ provides a standard exception handling mechanism/technique. We can detect errors by debugging.

- The purpose of the exception handling mechanism is to provide means to detect and report an exceptional circumstance or condition, so that appropriate action can be taken against it.
- When a function in a program detects an exceptional condition or situation, which represent this with exception object and in order to deal with the exceptional situation we throw the exception.
- This passes control as well as the exception, to a designated block of code in a direct or indirect caller of the function that throw the exception, this block of code is known as handler.
- In a handler block we specify the types of exceptions that it may process.
- C++ exception handling mechanism uses three keywords namely **try**, **catch** and **throw**.
- In C++ exception handling is done through the following steps:

  Step 1 : Hit the exception (find the exception).

  Step 2 : throw the exception (report the error).

  Step 3 : catch the exception (receive the error information).

  Step 4 : Handle the exception (perform corrective measures or solution against the problems).
- The relationship of try, catch and throw exception handling keywords called the exception handling model is shown in Fig. 6.7.
- The keyword try is used to preface a block of statements (surrounded by braces) which may generate exceptions. This block of statements is known as try block. When an exception is detected, it is thrown using a throw statement in the try block. Catch blocks defined by the keyword catch, 'catches' the exception 'thrown' by the throw statement in the try block, and handles it appropriately.

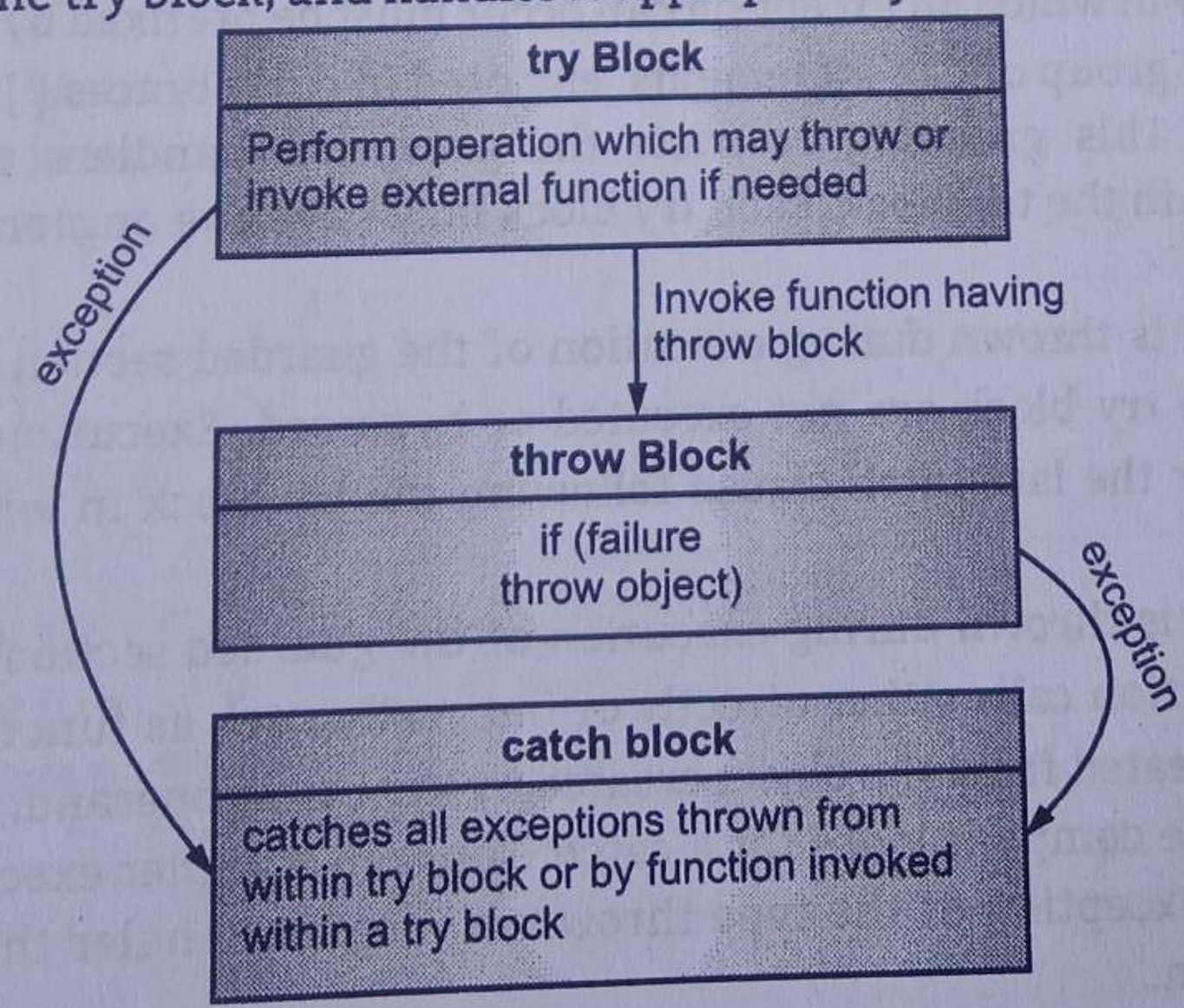


Fig. 6.7: Exception Handling Model

- The general **syntax** of code for this kind of relationship is shown below:

```
type function(arg list)    //Function with exception
{
    ........
    ........
    throw(object);                    //Throws exception
    ........
    ........
}
............
............
try
{
    ........
    ........                          //Invoke function here
    ........
}
catch(type arg)                       //Catches exception
{
    ........
    ........                          //Handles exception here
    ........
}
........
```

6.5.1 try Block

- A block of code in which an exception can occur must be prefixed by the keyword try.
- A try block is a group of C++ statements, enclosed in curly braces { }, that might cause an exception. This grouping restricts the exception handlers to the exceptions generated within the try block. Each try block may have one or more associated catch blocks.
- If no exception is thrown during execution of the guarded section, the catch clauses that follow the try block are not executed or bypassed. Execution continues at the statement after the last catch clause following the try block in which the exception was thrown.
- If an exception is thrown during execution of the guarded section or in any routine the guarded section calls either directly or indirectly such as functions, an exception object will be created from the object created by the throw operand.
- At this point, the compiler looks for a catch clause in a higher execution context that can handle an exception of the type thrown or a catch handler that can handle any type of exception.

- **Syntax:**

```
try
{
    ........  // statements in which exceptions may be thrown
    .........
    .........
}
```

6.5.2 catch Block

- The try block must be followed immediately by a handler, which is a catch block.
- A catch block is a group of C++ statements that are used to handle a specific thrown exception.
- One or more catch blocks, or handlers, should be placed after each try block. A catch block is specified by:
  1. The keyword catch.
  2. A catch parameter, enclosed in parentheses (), which corresponds to a specific type of exception that may be thrown by the try block.
  3. A group of statements, enclosed in curly braces { }, whose purpose is to handle the exception. [S-18]
- **Syntax for try and catch:**

```
try
{
    ...............................        //try block
    throw exception                        //detects and throws exception
    ...............................
}
catch(type1 arg)
{ //catch block }
catch(type2 arg)
{ //catch block }
    .
    .
    .
catch(typeN arg)
{ //catch block }
```

Program 6.6: Program for simple try-catch mechanism.

```
#include<iostream>
//using namespace std;
int main()
{
    cout<<"start\n";
```

```
try
{
    cout<<"try block \n";
    throw 10;     //throw an error
    cout<<"This will not work";
}
catch(int i)
{
    cout<<"exception caught, Number is:";
    cout<<i<<"\n";
}
    cout <<"end";
    return 0;
}
```

Output:

```
start
try block
exception caught, Number is:10
end
```

- In this above Program 6.6:
  - In try block, only two statements will execute the first cout statement and second throw. Once, the exception is thrown, control passes to the catch block and try block is terminated.
  - When the control passes to catch block, the statements in the catch block are executed.
  - Here, exception is integer exception.
  - The statement using namespace std is used.

Program 6.7: Simple program for try-catch statement.

```
#include<iostream>
using namespace std;
int main()
{
    int a, b;
    cin>>a>>b;
    int s = a/b;
    try
    {
        if(s!= 0)
        {
            cout<<"Result is="<<a/s<<'\n';
        }
```

```
    else
        {
            throw(s);        //throws integer exception
        }
    }
    catch(int i)
    {
        cout<<"exception caught:s="<<s<<endl;
    }
    return 0;
}
```

- Above program detects a division by zero error in try block. The program will not abort due to catch block.
- The exception is thrown using the object s.
- Catch statement contains integer type argument, catches the exception.
- If we change the argument in catch statement as double, then the exception will not be caught and abnormal termination occur.

Multiple Catch Statements:

- We can have more than one catch associated with a try. When the exception is thrown, the exception handler is searched in order for an appropriate match. When the match is found that catch block is executed. When no match is found, the program is terminated.
- **Syntax:**

```
try
{
    ................    // C++ statements
    ................
}
catch(type1 arg)
{
    ................    // catch block1
    ................
}
catch(type2 arg)
{
    ................    // catch block2
    ................
}
catch(typen arg)
{
    ................    // catch blockn
    ................
}
```

Program 6.8: Program for multiple catch statements.

```
#include<iostream>
using namespace std;
void func(int x)
{
    try
    {
        if(x==1) throw x;                //int
        else
        if(x==0) throw 'x';              //char
        else
        if(x== -1) throw 1.5;            //double
        cout<<"try ends:\n";
    }
    catch(char ch)
    {
        cout<<"caught character:\n"<<ch<<'\n';
    }
    catch(int i)
    {
        cout<<"caught integer:\n"<<i<<"\n";
    }
    catch(double d)
    {
        cout<<"caught double:\n"<<d<<"\n";
    }
    cout<<"end of catch block";
}
int main()
{
    cout<<"start \n";
    func(1);
    func(0);
    func(-1);
    func(2);
    return 0;
}
```

Output:

```
start
caught integer: 1
end of catch block
caught character: 0
end of catch block
caught integer: -1
end of catch block
try ends:
end of catch block
```

- In above program, when x = 1, it throws integer exception. This matches with the second catch block, therefore the second catch will executed. When x = 0, it throws character exception, therefore first catch block is executed. When x = –1, it throws double exception hence 3rd catch block is executed. Finally, when x = 2, program terminates (no. exception thrown).

6.5.3 throw Statement

- throw statement is used to throw an exception (an errors) inside of a try block to a subsequent exception handler.
- A throw statement is specified with the keyword 'throw' and an assignment expression.
- When an exception that is desired to be handled is detected, it is thrown using the throw statement.
- **Syntax:**

```
throw (exception);  or
throw exception;    or
throw;
```

- The throw statement is used to throw an exception and its value to a matching catch exception handler. A regular throw consists of the keyword throw and an expression. The result type of the expression determines which catch block receives control.
- Within a catch block, the current exception and value may be re-thrown simply by specifying the throw keyword alone that is without the expression.
- The throw is syntactically similar to the operand of a return statement but here, it returns to the catch handler.
- An exception can be thrown from a statement that is outside the try block but within a function that is called from within the try block.

Program 6.9: Program for throw statement.

```
#include<iostream>
using namespace std;
void funct(int t)
{
    cout<<"Inside function:"<<t<<'\n';
    if(t) throw t;
}
int main()
{
    cout<<"start\n";
    try
    {
    cout<<"Inside try \n";
        funct(0);
        funct(1);
```

```
        funct(2);
    }
    catch(int i)
    {
        cout<<"exception caught:"<<i<<"\n";
    }
    cout<<"end";
    return 0;
}
```

Output:

```
start
Inside try
Inside function:0
Inside function:1
exception caught:1
end
```

try-catch inside a Function:

- When try block is inside a function, then each time the function is entered, the exception handling relative to that function is reset.

Program 6.10: Program for try-catch block inside a function.

```
#include<iostream>
using namespace std;
void func(int x)
{
    try
    {
        if(x) throw x;
    }
    catch(int i)
    {
    cout<<"exception caught:"<< i <<"\n";
    }
}
int main()
{
    cout<<"start"<<"\n";
    func(1);                //for every function call
    func(2);                //exception handling is reset
    func(0);
    func(3);
    return 0;
}
```

Output:

```
start
exception caught:1
exception caught:2
exception caught:3
```

Summary

- A file is a collection of related or associated data on a particular area of the storage device. Text files and binary files are the two different types of files.
- The data in a text file is stored in the form of characters, i.e. in ASCII code. In binary files the data is stored in terms of a sequence of bytes, i.e., in 0's and 1's.
- File handling concept in C++ language is used for store a data permanently in computer. Using file handling we can store the data in secondary memory (hard disk).
- The fstream class inherits all the functions from istream and ostream classes through iostream class defined inside iostream.h file. Also fstream inherits from the ios class, which is also the base class for istream and ostream classes.
- The ifstream class is derived from the istream class. Therefore, all the functions of ifstream class can operate on fstream class.
- A file can be opened in two ways by using the constructor function of a class and using the member function open() of the class.
- The function open() does the following:
  - It prepares the file for input or output operation.
  - It places the file pointer at the appropriate location in the file depending upon the mode in which file has been opened.
- The close() function closes all the opened stream files.
- The error-handling mechanism of C++ is generally referred to as exception handling. It provides a straightforward mechanism for adding reliable error-handling mechanism in a program.
- Exception occurs when program tries to execute abnormal instruction.
- C++ exception handling mechanism is built upon three keywords i.e., try, catch, and throw.
- A try block in a C++ program identifies a block of code for which particular exceptions will be activated. It is followed by one or more catch blocks.
- A program throws an exception when a problem shows up. This is done using a throw keyword.
- The exception handler is indicated by the catch keyword. The handler must be used immediately after the try-block.
- The catch keyword indicates the catching of an exception.

Check Your Understanding

1. The stream that supplies data to a program is called _____ stream.
   (a) input (b) output
   (c) I/O (d) none of these
2. The stream that receives data from the program is called _____ stream.
   (a) input (b) output
   (c) I/O (d) none of these
3. Open() function is used to open _____ files which use the same stream objects.
   (a) multiple (b) console
   (c) print (d) none of these
4. _____ function is used to reach end of file.
   (a) exit() (b) close()
   (c) eof() (d) both (a) & (b)
5. The ..... function closes all the opened stream files.
   (a) exit() (b) close()
   (c) eof() (d) both (a) & (c)
6. An exception is thrown using _____ keyword.
   (a) throws (b) throw
   (c) threw (d) Thrown
7. Exception handlers are declared with ..... keyword.
   (a) try (b) catch
   (c) throw (d) finally
8. The try block contains the statements that _____.
   (a) throws exceptions (b) catch exceptions
   (c) correct exception (d) all of these
9. The code of statements which may cause abnormal termination of the program should be written under _____ block.
   (a) Try (b) catch
   (c) Finally (d) none of these
10. An exception is _____ error.
   (a) syntax (b) logical
   (c) run-time (d) none of these

Answers

1. (a)	2. (b)	3. (a)	4. (c)	5. (b)	6. (b)	7. (b)	8. (a)	9. (a)	10. (c)

Practice Questions

Q.I Answer the following questions in short:

1. What is the purpose of input stream and output stream?
2. What are the types of files?
3. Enlist various file operations.
4. What is an exception?
5. How an exception handled in C++?
6. What is use of catch(...) exception handler?

Q.II Answer the following questions:

1. With the help of diagram explain file operations.
2. Explain various classes available for file Operations.
3. Differentiate between opening a file with a Constructor function and opening a file with open( ) function.
4. What is a File Mode? Describe the various file mode options available.
5. Write a C++ program that reads a text file and creates another file that is identical except that every sequence of consecutive blank spaces is replaced by a single space.
6. Write a C++ program that will create a data file containing the list of telephone numbers and name. Use a class object to store each set of data.
7. Write a menu-driven program that will allow us to create a file with following fields or information:
   (a) book-name
   (b) code
   (c) author-name
   (d) price and implement the following tasks
   (e) add a record
   (f) modify a record
   (g) search a record.
8. Write a C++ program to count number of vowels in a text file.
9. Write a C++ program to count number of characters and word from text file.
10. Explain simple try and catch mechanism with an example.
11. When do we use multiple catch handler?
12. Explain exception handling mechanism diagrammatically.
13. Explain the significance of: (i) throw (ii) catch (iii) try
14. Write a program which uses multiple catch.

Q.III Define terms:

1. File
2. Stream
3. Input stream
4. Output stream

Previous Exam Questions

Summer 2018

1. The try block contain the statements that _____. [1 M]
   (i) throws exception
   (ii) catch exceptions
   (iii) correct exception
   (iv) all of these

Ans. Refer to Section 6.5.1.

2. Explain hierarchy of file stream classes with proper diagram. [3 M]

Ans. Refer to Section 6.2.

3. Explain simple try & catch mechanism with example. [4 M]

Ans. Refer to Section 6.5.2.

4. Write a program for opening two files with same stream & storing the content in it. [3 M]

Ans. Refer to Program 6.2.

Winter 2018

1. Give any two ways to check the success or failure of a file open operation. [1 M]

Ans. Refer to Section 6.3.1.

2. Explain the use of the following: [4 M]
   (i) Try block
   (ii) Throw statement
   (iii) Catch block.

Ans. Refer to Section 6.5.

3. How to handle an exception in C++? [3 M]

Ans. Refer to Section 6.5.

Summer 2019

1. _____ function is used to check whether the file pointer has reached at end of file or not.
   (a) eof( )
   (b) close( )
   (c) open( )
   (d) exit( ) [1 M]

Ans. (a) eof( )

2. Give purpose of FilebuF Class From "Fstream.h". [1 M]

Ans. Refer to Section 6.2.

3. Explain various file opening modes in C++. [4 M]

Ans. Refer to Section 6.3.1.2.