



**YASHAVANT
KANETKAR**



Highlights Core Features Like Encapsulation,
Polymorphism, Inheritance, Virtual Functions, Templates,
Exception Handling, STL and more...

Let Us **C++**

If you liked Kanetkar's Let Us C, You would like Let Us C++ more.



Let Us

C++

Third Edition

Yashavant Kanetkar



THIRD REVISED AND UPDATED EDITION 2019

Copyright © BPB Publications, INDIA

ISBN: 9789388176644

All Rights Reserved. No part of this publication can be stored in a retrieval system or reproduced in any form or by any means without the prior written permission of the publishers.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The Author and Publisher of this book have tried their best to ensure that the programmes, procedures and functions described in the book are correct. However, the author and the publishers make no warranty of any kind, expressed or implied, with regard to these programmes or the documentation contained in the book. The author and publisher shall not be liable in any event of any damages, incidental or consequential, in connection with, or arising out of the furnishing, performance or use of these programmes, procedures and functions. Product name mentioned are used for identification purposes only and may be trademarks of their respective companies.

All trademarks referred to in the book are acknowledged as properties of their respective owners.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

Published by Manish Jain for BPB Publications, 20, Ansari Road, Darya Ganj, New Delhi- 110002 and Printed him at Repro India Pvt Ltd, Mumbai

Dedicated to

Nalinee & Prabhakar Kanetkar...

About the Author

Through his books and Quest Video Courseware DVDs on C, C++, Data Structures, VC++, .NET, Embedded Systems, etc.

Yashavant Kanetkar has created, molded and groomed lacs of IT careers in the last two and half decades. Yashavant's books and Quest DVDs have made a significant contribution in creating top-notch IT manpower in India and abroad.

Yashavant's books are globally recognized and millions of students / professionals have benefitted from them.

Yashavant's books have been translated into Hindi, Gujarati, Japanese, Korean and Chinese languages. Many of his books are published in India, USA, Japan, Singapore, Korea and China.

Yashavant is a much sought after speaker in the IT field and has conducted seminars/workshops at TedEx, IITs, RECs and global software companies.

Yashavant has been honored with the prestigious “Distinguished Alumnus Award” by IIT Kanpur for his entrepreneurial, professional and academic excellence. This award was given to top 50 alumni of IIT Kanpur who have

made significant contribution towards their profession and betterment of society in the last 50 years.

In recognition of his immense contribution to IT education in India, he has been awarded the “Best .NET Technical Contributor” and “Most Valuable Professional” awards by Microsoft for 5 successive years.

Yashavant holds a BE from VJTI Mumbai and M.Tech. from IIT Kanpur. Yashavant's current affiliations include being a Director of KICIT Pvt. Ltd. and KSET Pvt. Ltd.

Acknowledgments

The joy of creativity! Probably, that's why I am drawn to writing. I believe that at some stage of book writing the book starts taking a life of its own. Once it does that, the author just has to kneel down, claw at the earth, overturn the soil and pray for the rain. The book is capable of unfolding itself. There is no need for the author to take credit for it.

This has been one long book project and naturally many people got associated with it. Any bouquets for the merit in this book should go to their door. Any brickbats, I am ready to catch myself.

Shakeel Ali helped in two ways—We together straightened out poorly written or confusing text. He alone pulled, pushed and shoved the code in this book till he got it perfectly right.

Nandita Hingwe solved all the exercises in the book and made sure that the questions make sense to the reader. Figures in this book too, are all hers.

I am grateful to all the students who attended my C++ lectures and seminars and helped me to improve my

understanding of numerous C++ concepts. More than anything else, I learnt from them that if you are open to new ideas, your circle of friends can go much beyond where your car can drive.

Over the year I have become a little choosy about the cover of a book. The artist in Vinay Indoria is so adept at his unique skills that he creates only one artwork and I choose one out of one. Many thanks to him, for yet another wonderful cover.

One person, who has steadfastly stood by me in all my book writing efforts, is my wife Seema. This book has been no different. She has helped me in more ways than what my words can capture.

Contents

1 Intro to OOP

The Beginning...

Structured Programming

Object Oriented Programming

Characteristics of Object Oriented Programming

Exercise

KanNotes

2 Graduating to C++

Comments

Input / Output in C++

Flexible Declarations

Flexible Initializations

Inferring Types

union and *enum* Syntax

Anonymous and

Typecasting

void Pointers

The `::` Operator

References

The `const` Qualifier

bool Data Type

Exercise

[KanNotes](#)

[3 Functions](#)

[Strict Prototype Checking](#)

[Default Values for Function Arguments](#)

[Function Overloading](#)

[Operator Overloading](#)

[Inline Functions](#)

[New Return Type Syntax](#)

[Instance, virtual and friend Functions](#)

[Exercise](#)

[KanNotes](#)

[4 Classes and Objects](#)

[Structures and Classes](#)

[Classes and Constructors](#)

[Destructors](#)

[A Complex Class](#)

[The this Pointer](#)

[Overloading Unary Operators](#)

[Objects and Memory](#)

[Structures and Classes Revisited](#)

[Ideal Program Organization](#)

[Exercise](#)

[KanNotes](#)

5 Class Intricacies

Static and Dynamic Memory Allocation

Static Members

The Multi-purpose const

Overloaded = and Copy Constructor

Data Conversion

Exercise

KanNotes

6 Inheritance

Inheritance

Another Inheritance Example

Uses of Inheritance

Constructors in Inheritance

Types of Inheritance

A Word of Caution

Incremental Development

Exercise

KanNotes

7 Polymorphism

Virtual Functions

Pure Virtual Functions

Abstract Class

Function Binding

Virtual Functions under the Hood

[Why use Virtual Functions?](#)

[Object Slicing](#)

[Virtual Destructors](#)

[Virtual Base Classes](#)

[Exercise](#)

[KanNotes](#)

[8 Input / Output in C++](#)

[Expectations from an I/O System](#)

[C++ Streams Solution](#)

[Ready-made Stream Objects](#)

[The *iostream* Library](#)

[The *istream* Class](#)

[The *ostream* Class](#)

[The *iostream* Class](#)

[Stream Manipulators](#)

[User-defined Manipulators](#)

[User-defined Manipulators with Arguments](#)

[File I/O with Streams](#)

[Character I/O](#)

[A Filecopy Program](#)

[Line I/O](#)

[Record I/O](#)

[Random Access](#)

[File Opening Modes](#)

[String Streams](#)

[Object I/O](#)

[Serialization](#)

[Errors Handling during I/O](#)

[Interaction with File System](#)

[Exercise](#)

[KanNotes](#)

[9 Advanced Features of C++](#)

[Containership](#)

[friend Functions and friend Classes](#)

[One More Use of friend Function](#)

[A Word of Caution](#)

[The explicit Keyword](#)

[The mutable Keyword](#)

[Namespaces](#)

[Using a Namespace](#)

[RTTI](#)

[Typecasting in C++](#)

[Pointers to Members](#)

[Exercise](#)

[KanNotes](#)

[10 Templates](#)

[Function Templates](#)

[Function Templates for User-defined Types](#)

[One More Function Template](#)

[Multiple Argument Types](#)

[Templates versus Macros](#)

[A Template-based Sort](#)

[Class Templates](#)

[A Linked List Class Template](#)

[Tips about Templates](#)

[Variadic Templates](#)

[Application of Templates](#)

[Exercise](#)

[KanNotes](#)

11 Exception Handling

[Exception Handling in C++](#)

[Using Ready-made Exception Classes](#)

[Using User-defined Exception Class](#)

[A Few Tips](#)

[Exception Specification](#)

[Unhandled Exceptions](#)

[Smart Pointers and Dynamic Containers](#)

[Exercise](#)

[KanNotes](#)

12 Standard Template Library

[Standard Template Library](#)

[Components of STL](#)

[vector Container](#)

[Vector of Point Objects](#)

[list Container](#)

[Sets and Multi-sets](#)

[Maps and Multi-maps](#)

[stack Container](#)

[queue Container](#)

[Exercise](#)

[KanNotes](#)

[Index](#)

Preface to Third Edition

Things have changed a lot since I wrote the first edition of this book, about a decade ago. That time C++ was the new kid on the block. Today it is a mature, powerful and a popular choice for programmer's community. I wanted to reflect this change in this new edition.

So I took each program in the book, checked whether it works with popular compilers of today like Visual Studio and gcc, modified them when they didn't and finally I am presenting them to you. The text too has been modified to accommodate the changes made in the programs.

Most programmers migrate from C to C++. One of the biggest mistakes they commit is that most of the time they are doing C programming with a C++ compiler. This is one aspect that I wanted to change, because if this is not done you would never be able to exploit the real power that C++ lays at your door-steps. Hence, in the very first chapter itself I have discussed how one should *think* about any problem in C++ way. I am hopeful you would appreciate this approach and benefit from it.

When I talk to participants in training rooms, seminars, or workshops one question that is asked frequently is—‘how do I brush up all aspects of C++ programming before an all-important viva-voce or an interview?’ I have answered that by creating a section called ‘KanNotes’ at the end of each chapter. I am hopeful that you like this idea.

In this edition I have significantly improved several chapters, especially those on Inheritance, Polymorphism, Templates and Exception Handling. Since most worth-while C++ programs make use of Standard Template Library to handle collections, I have added one full chapter on it.

C++ has undergone lot of transitions from C++98 to C++11 to C++14 to C++17. At the time of this writing, most of the C++ compilers have fully implemented features of C++14. Same is not true about C++17. Hence I have tested all the programs in this book with C++14 compilers.

All in all, I would say that this is a C++ book written to help you learn this wonderful language with latest development tools in the right perspective. All the best and I hope you would enjoy reading it.

Preface to First Edition

Completing this book is one of the hardest things that I have ever done. It took me almost two years to get it into the form you are reading.

Every time I read the earlier draft, I had to almost always rewrite the whole thing. Two reasons for this—one C++ is abstract, second it is complex. Most C++ programmers are former C programmers. I too migrated to C++ from C. Hence, unless I could contrast any new C++ feature with how it was being done in C and tell readers how it could be done better in C++, I felt I would be failing in my duty.

C++ is decorated (?) with a lot of bombastic jargon. One of my aims here was to keep this jargon at bay and concentrate on the underlying concepts instead. At most places, I have tried to show how things work and more importantly, why do they work that way.

If you ask me to name the most important characteristic of this book, I would say Be it the code or the text, I have tried to make it as simple as I could. As far as the code is concerned, I wanted to present simple examples that can be easily edited, compiled and run. My goal was not to

demonstrate how good a programmer I am by adding gloss to these programs, but to illustrate specific programming concepts.

You will also notice that very few programming examples in this book are code fragments. I have found that a program that can actually compile and run, no matter how simple it is, helps improve one's understanding of a subject a great deal more, than just code fragments.

I have found that simple exercises are exceptionally useful to complete the reader's understanding of a topic. So you will find one at the end of each chapter.

More than anything else, I have tried to design the book for a programmer struggling with a new and complex programming language. I have poured my best efforts into these pages. I trust you would find the book useful.

All the best and happy programming!

Yashavant Kanetkar

Introduction To OOP

Object Oriented programming languages like C++ and Java have emerged as the top choice of programmers to create complex applications for the modern digital world. Before we dive into the nitty-gritties of C++ language, it is important to understand why programmers prefer OOPs and C++

The Beginning...

Structured Programming

Object Oriented Programming

Characteristics of Object Oriented Languages

Objects

Classes

Encapsulation

Data hiding

Inheritance

Polymorphism

Containership

Templates

Exception handling

Reusability

Exercise

KanNotes

What is object-oriented programming (OOP)? This question is little difficult to answer because the software industry has a fascination for terminologies and catch words. Not long ago, words like “artificial intelligence”, “WAP” and “Java” were used as if they were to offer a path to heaven. The same overuse seems to be happening to the phrase “object-oriented”. Since it has been proven that object-oriented techniques offer a way to write better programs, everybody seems to be slapping the label “object-oriented” on their software products.

Hence it is important for us to understand what is OOP, why do we need it, what does it do that traditional languages like C, Pascal and Basic don't and what are the principles behind OOP. This chapter addresses these issues and provides an overview of the features to be discussed in the rest of the book. What we say here will necessarily be general and brief. Don't worry if you don't catch everything in this chapter on the first pass; OOP is a bit complex and understanding it takes time. We will be going over these features again in subsequent chapters. There's lot of ground to cover here, so let's get started.

The purpose of a programming language is to express the solution to a problem with the help of an algorithm (step by step procedure). The success of the solution depends on how the solution models (represents) the problem. Different approaches have evolved over the years to model solutions to problems. The primary amongst them are Structured programming model (also called Procedural programming model) and Object-oriented programming model. Of late, the structured programming model is being replaced by object-oriented programming model. To understand these models we need to begin by taking a peek at the history of programming models.

The Beginning...

The earliest computers were programmed in binary. Mechanical switches were used to load programs. With the advent of mass storage devices and larger and cheaper computer memories, the first high-level computer programming languages came into existence. With their arrival, instead of thinking in terms of bits and bytes, programmers could write a series of English-like instructions that a compiler could translate into the binary language of computers.

These languages were simple in design and easy to use because programs at that time were primarily concerned with relatively simple tasks like calculations. As a result, programs were pretty short, limited to about a few hundred lines of instructions.

As the computers' capacity and capability increased, so also did the ability to develop more complex computer programs. However, the earlier programming languages were found wanting in performing the complex programming tasks. These languages suffered from the following limitations:

There were no facilities to reuse existing program code. Wherever the same piece code was required, it was simply duplicated.

The control of execution within a program was transferred via the dangerous **goto** statement. As a result, there was too much jumping around in the program, often without any clear indication of how, where and why the control is flowing.

All variables in the program were global. Tracking down spurious changes in global data in long convoluted programs was a very tedious job.

Writing, understanding and maintaining long programs became a programmer's nightmare.

In short, we can call this methodology of developing programs as **Unstructured** programming.

Structured Programming

To overcome the limitations mentioned above, a quest began to develop new languages with new features that would help to create more sophisticated applications. The breakthrough occurred in late 1960's and early 1970's with the introduction of structured programming. The long programs that the programmer found difficult to comprehend could now be broken down into smaller units of few hundred statements. Functions/subroutines/procedures were introduced in these languages to make the programs more comprehensible to their human creators. A program was now divided into functions, with each function having a clearly defined purpose. How structured programming overcame the limitations experienced in unstructured programming is given below.

Reuse of existing program code - Wherever the same piece code is required at multiple places in a program, the function containing that code is used. As a result, there is no need to repeat the same code at multiple places.

Excessive use of **goto** statement - The excessive use could be minimized with the introduction of powerful control instructions that could transfer the control within the program in an easy-to-understand manner.

Unexpected changes in global variables - With introduction of functions, need for global variables got minimized.

Complexity of programs - Complexity became more manageable as structured programming permitted better organization of the program.

A structured program is built by breaking down a solution into smaller pieces (also called as divide and conquer technique) that then become functions within that program. Each function can have its local variables and logic. The execution begins with one function and then all other functions are called directly or indirectly from this function. This is shown in [Figure](#)

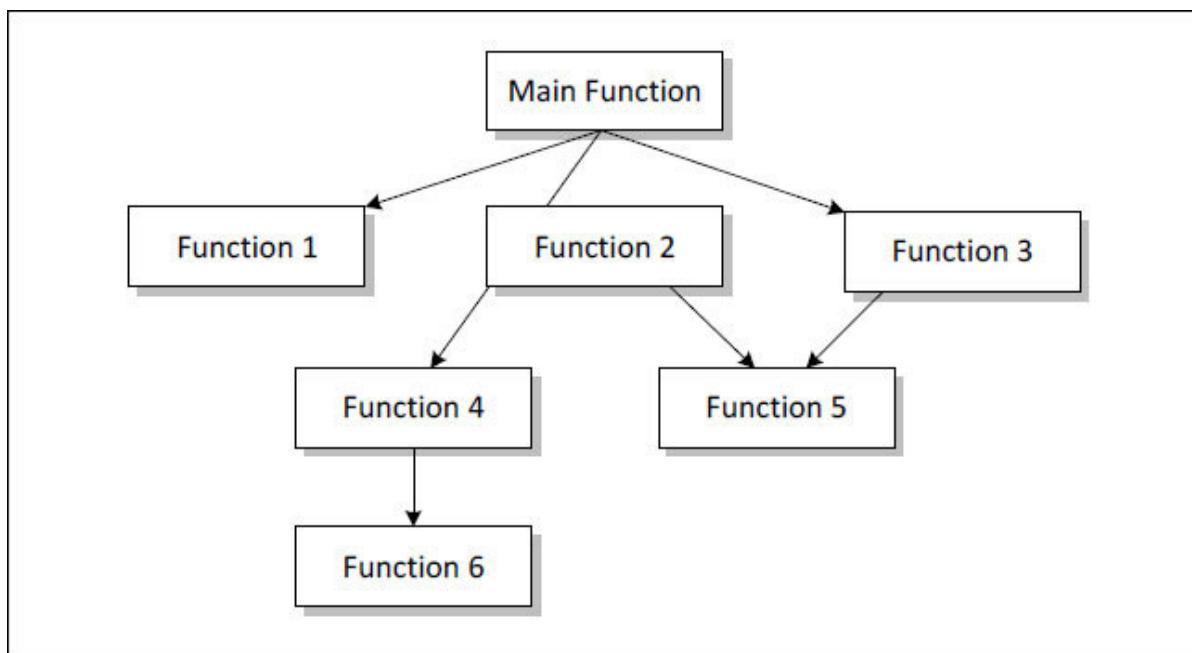


Figure Structured programming.

In structured programming, there is less need of global variables, which are now replaced by local variables that have a smaller and more controllable scope. Information is passed between functions using parameters and functions can have local variables that cannot be accessed outside the function's scope.

By isolating processes within functions, a structured program minimizes the chance that one function will affect another. This also makes it easier to locate problems, if any. Structured programming helps you to write cleaner code and maintain control over each function. All this makes the development and maintenance of code easier as well as efficient.

A new concept came into existence with structured programming — Abstraction permits the programmer to look at something without being concerned with its internal details. In a structured program, it is enough to know which task is performed by function. It does not matter to the programmer *how* that task is performed so long as the function does it reliably. This is called functional abstraction and is the corner-stone of structured programming.

Structured programming dominated the software world for almost two decades—from 1970 to 1990. With the constant improvement in the hardware and increasing demands from the users for feature-rich programs, the complexity of programs increased multi-

fold and that's the time structured programming approach started showing signs of strain. This occurred because of the following weaknesses in the structured programming model:

The primary components of structured programming—functions and data structures—didn't model the real world problems in a natural way.

Mechanisms to reuse existing code were limited.

Maintaining, debugging and upgrading large programs were a difficult task.

The solution to these limitations is discussed in the next section.

Object Oriented Programming

The real-world problems and their solutions are not organized into values and procedures separate from one another. Problem solvers do not perceive the world that way. They deal with their problem domains by concentrating on the objects and letting the characteristics of those objects determine the procedures to apply to them. To build a house, grow a tomato, or repair an engine, first you think about the object and its purpose and behavior. Then you select your tools and procedures. The solution fits the problem.

Thus the world is object-oriented, and the object-oriented programming methodology expresses computer programs in ways that model how people perceive the world. Because programmers are people, it is only natural that their approach to the work of the world reflects their view of the world itself.

The object-oriented methodology is built on the foundation laid by the structured programming concepts and data abstraction. Data abstraction does for data what functional abstraction does for operations. With data abstraction, data structures can be used without having to be concerned about the exact details of implementation. For example, floating-point numbers are abstracted in programming languages. You are not required to

know how a floating-point number is represented in binary while assigning a value to it. Likewise, you are not bothered how binary multiplication takes place while multiplying floating-point numbers. Abstraction for floating-point numbers has existed in programming languages since long. However, it is only recently that languages have been developed that let you define your own abstract data types.

The fundamental change in OOP is that a program is designed around the data being operated upon, rather than around the operations themselves. This is to be expected once we appreciate that the very purpose of the program is to access or manipulate data. The basic idea behind object-oriented language is to combine into a single unit, both, the data and the functions that operate on the data. Such a unit is called an object.

An object's functions, called member functions or methods in C++, typically provide the only way to access its data. If you want to access a data item in an object, you call a member function in the object. It will read the item and return the value to you. You can't access the data directly.

If you want to modify the data in an object, you know exactly which functions interact with it—the member functions in the object. No other functions can access the data. This simplifies writing, debugging, and maintaining the program.

A C++ program typically consists of a number of objects which communicate with each other by calling one another's member functions. The organization of a C++ program is shown in [Figure](#)

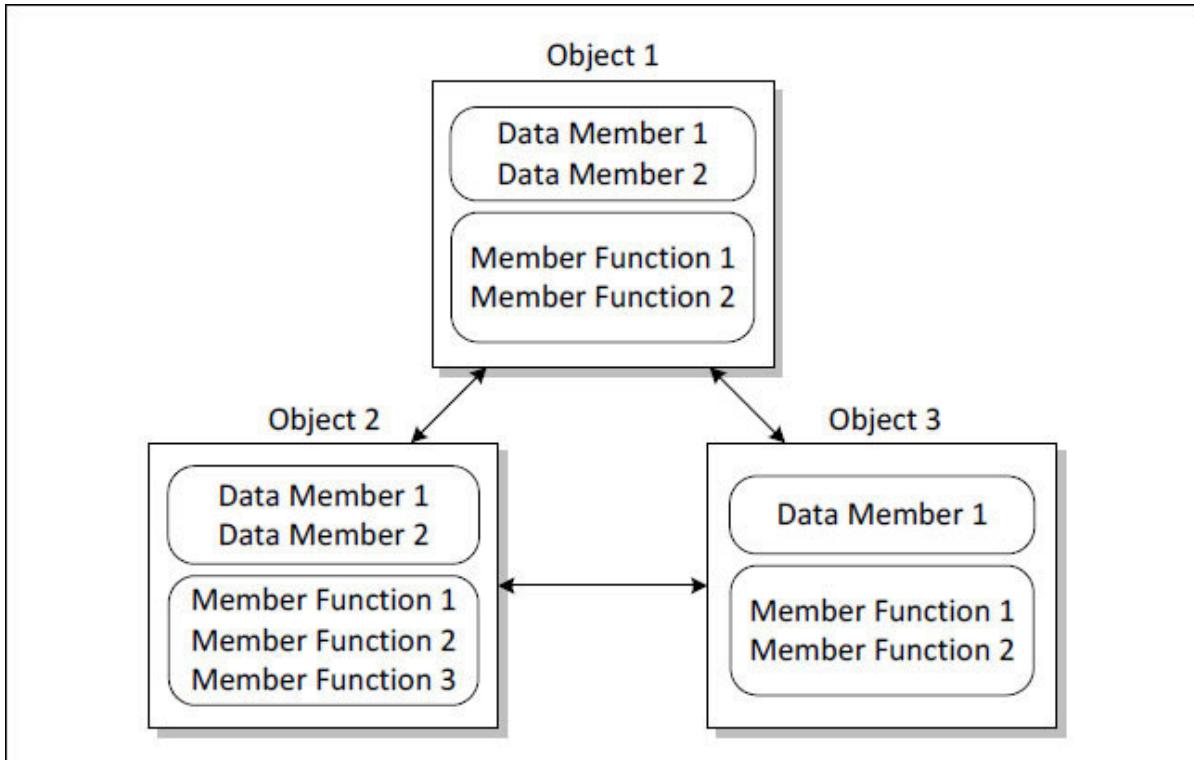


Figure *Object-oriented programming.*

When you approach a programming problem in an object-oriented manner, you no longer ask how the problem will be divided into functions, but rather how it will be divided into objects. Thinking in terms of objects, rather than functions, has a surprisingly helpful effect on how easily programs can be designed. This results from the close match between objects in the programming world and objects in the real world.

The match between programming objects and real world objects is the happy result of combining data and functions. The resulting objects offer a revolution in program design. No such close match between programming constructs and the concepts being modeled exists in a procedural language.

There is more to OOP than just binding the data and functions together. Given below are some of the new concepts introduced in OOP:

Encapsulation

Data hiding

Inheritance

Containership

Polymorphism

Templates

Exception handling

Reusability

Don't get daunted by this list of new features. They are explained in brief in the next section. A detailed explanation of each of these concepts is given in chapters to follow.

Characteristics of Object-Oriented Languages

Object-oriented programming uses a vocabulary that is unfamiliar to the procedural programmer. Let us now briefly examine this vocabulary with regards to the major elements of object-oriented languages.

OBJECTS

In structured programming a problem is approached by dividing it into functions. Unlike this, in object-oriented programming the problem is divided into objects. Thinking in terms of objects rather than functions makes the designing of program easier. Following are few candidates that can be treated as objects in respective situations:

- Employees in a Payroll processing system
- Data structures like linked lists, stacks, queues etc.
- GUI elements like windows, menus, icons etc.
- Hardware devices like disk drive, keyboard, printer, etc.

- Various elements in computer games like cannons, guns, animals, etc.
- Customers, sales persons in a sales tracking system
- Computers in a network model

CLASSES

Most languages offer primitive data types like **long** and Their data representation and response to arithmetic, assignment and relational operators are defined as part of the language. However, not all the information about real world objects can be represented using these limited built-in data types. The programmer often needs to create his own data types by defining a **class** for it.

For example, there can be a user-defined data type to represent dates. The compiler and the computer do not know about dates. Programmers have to define the behavior of dates by designing a **Date** class. This class expresses the format of a date and the operations that can be performed on it. The way, we can declare many variables of the primitive type we can define many objects of the **Date** class. A class serves as a blueprint or a plan or a template. It specifies what data and what functions will be

included in objects of that type. Defining a class doesn't create any objects, just as the mere existence of a type **int** doesn't create any variables.

ENCAPSULATION

The packaging of data values and functions within an object is referred to as encapsulation. For example, suppose we define a user-defined type (class) called **Box** and build objects **b1** and **b2** from this user-defined type. Then these objects may encapsulate data members **breadth** and **height** of the box and member functions **calcSurfaceArea()** and

Another example could be that of objects of a user-defined class called **Objects**. Objects of this class may encapsulate data like **weight** and functions like etc.

DATA HIDING

C++ offers mechanisms through which we can permit or deny access to the data members and member functions in an object. Usually, data is kept inaccessible from outside the object. It can be accessed only through the member functions present in the object. This process of making the data inaccessible from outside the object is often called data hiding.

Don't feel that this is a restrictive approach—what is the use of data members that cannot be accessed or modified from outside the object? If you wish to access/modify the data you can always do it systematically through the member functions of the object. This avoids inadvertent access or manipulation of data and the resulting consequences.

INHERITANCE

OOP permits you to create your own data types (classes) just like the types built into the language. However, unlike the built-in data types, the user-defined classes can use other classes as building blocks. Using a concept called new classes can be built on top of the old ones. The new class referred to as a **derived** class, can inherit the data and functions of the original, or the **base** class. The new class can add its own data elements and functions in addition to those it inherits from its base class.

For example, we can build a set of classes that describe a library of publications. There are two primary types of publications—periodicals and books. We can create a general **Publication** class by defining data items for the publisher name, the number of pages and the accession number. Publications can be retrieved, stored and read. This can be done through functions of **Publication** class.

Next we can define two classes named **Periodical** and Both these classes can be derived from the base class This is natural because a periodical as well as a book has properties publisher name, number of pages and the accession number.

In addition to this, a periodical also has a volume number and issue number and contains articles written by different authors. Data items for these should be included in the definition of the **Periodical** class. The **Periodical** class will also need a function, subscribe.

Data items for the **Book** class will include the name of its author, a cover type (hard or soft) and its ISBN number. This class will also have a function called As you can see, the **Book** class and the **Periodical** class share the characteristics of **Publication** class while having their own unique attributes. This entire scenario is depicted in [Figure](#)

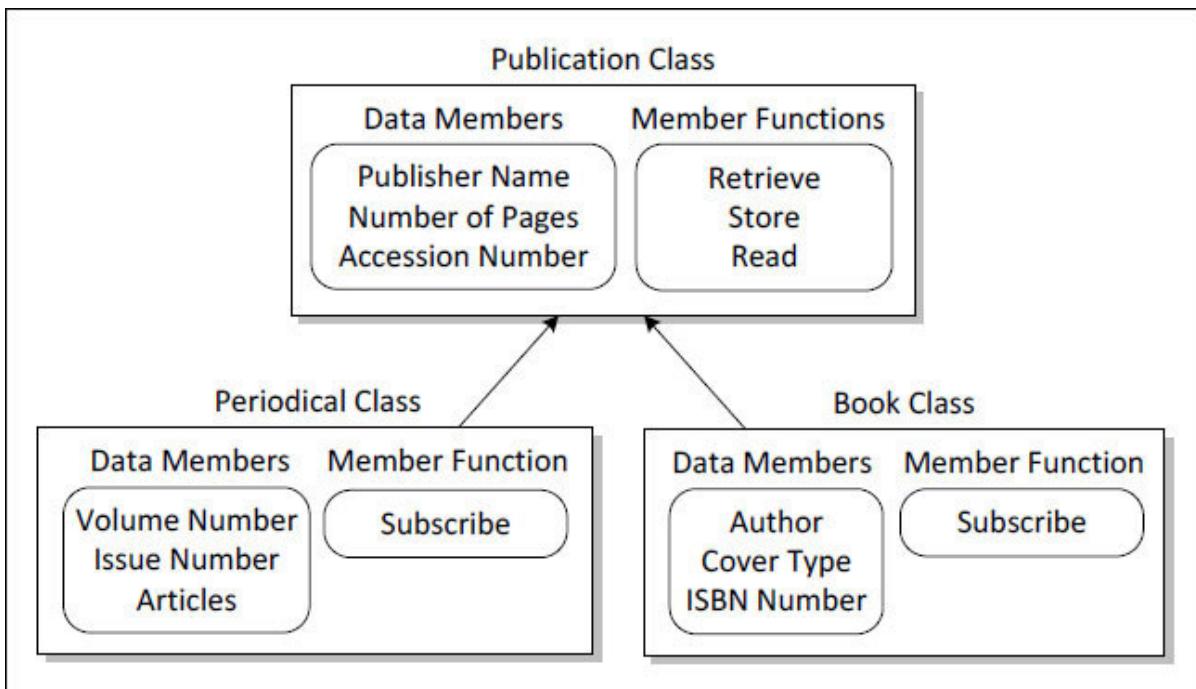


Figure Example of inheritance.

POLYMORPHISM

Extending the same example of the **Periodical** and let us now understand another important concept. Our base class, defines methods for storing and retrieving data. A periodical may be stored in a binder, while a book is usually placed on a shelf. Furthermore, the way to find a specific periodical is different from finding a book. Periodicals are located through a guide to periodical literature, while books are found using a card catalogue system. Based on this we can design a ‘find through periodical literature’ function for a periodical and a ‘find through card

'catalogue' function for a book. OOP provides an elegant facility called **polymorphism** to handle such situations.

In our example, the retrieval method for a periodical is different from the retrieval method for a book, even though the end result is same. Polymorphism permits us to define a function for retrieving a publication that can work for both periodicals and books. When a periodical is retrieved, the retrieve function that is specific to a periodical is used, but when a book is retrieved, the retrieve function associated with a book is used. The end result is that a single function name can be used for the same operation performed on related derived classes even if the implementation of that function varies from class to class. This concept of polymorphism (one thing with several distinct forms) can be extended even to operators, as we will see in later chapters.

CONTAINERSHIP

In a typical super-market each item on sale can be represented using a class. These items in turn belong to different categories like cosmetics, food, cold-drink, clothes, books, electronics, etc. Such relationships can be represented using containership. For example objects like cold- cream, face-wash, shampoo are contained inside a category object called cosmetics. You will be able to observe this containership relationship in many real-world problems.

TEMPLATES

You sometimes need several similar functions or classes in a program where the code differs only in the type of data that is processed. For example, you may need a set of **abs()** functions one that returns absolute value of an another that returns absolute value of a etc. In such situations, instead of defining and maintaining this set of functions, C++ permits us to use a feature called **Template** to create only one generic function. Then, depending upon the need, the compiler will generate specific versions of this function to work with different data types.

On similar lines you can imagine a generic templated class called From this generic class compiler can create specific classes—one to maintain a stack of integers, another to maintain a stack of floats, etc.

EXCEPTION HANDLING

A robust program must anticipate hardware/software errors or abnormal events that are likely to occur during its execution and handle them. For example, a hardware error may occur while accessing a file/database/printer and a software error may occur if the denominator has a value 0 while performing a division. These are abnormal situations and hence are aptly called exceptions.

C++ lets us handle these exceptions in object-oriented manner. For this, we create an exception object, pack it with relevant information about the error condition and then throw this exception object (using the `throw` keyword). This thrown exception can then be caught in another part of the code (using the `catch` keyword) and then dealt with suitably.

REUSABILITY

Object-oriented programs are built from reusable software components. Once a class is completed and tested, it can be distributed to other programmers for use in their own programs. This is called reusability. If those programmers want to add new features or change the existing ones, new classes can be derived from existing ones. The tried and tested capabilities of base classes do not need to be redeveloped. Programmers can devote time to writing new code instead of wasting time in rewriting existing code. This way software becomes easier to test, since programming errors can be isolated within the new code of derived classes.

For example, you might have written (or purchased from someone else) a class that creates a menu system. You are happy with the working of this class and you don't want to change it, but you want to add the capability of displaying help for each menu item. To do this, you simply create a new class that inherits all the capabilities of the existing one but adds help feature. This ease

with which existing software can be reused is a major benefit of OOP.

Exercise

[A] State whether the following statements are True or False:

Object oriented programming permits reusability of the existing code.

Languages earlier than procedural programming languages made use of only global variables.

In Object Oriented programming languages the only way to transfer control from one place in the program to another is by using the goto statement.

It is easier to write, understand and maintain programs if they use Object Oriented Programming model as compared to Structured Programming model.

As compared to procedures, data is not given enough importance in Procedural programming.

Structured programming model does not represent the real world problem as well as the Object Oriented programming model.

Object Oriented programming model permits functional abstraction as well as data abstraction.

A class permits us to build user-defined data types.

Objects are to classes as variables are to built-in data types.

Deriving a new class from an existing class promotes reusability of code.

Encapsulation facilitates a single function name to be used for the same operation performed on related derived classes.

In polymorphism even though the function names are same, their implementation may vary from class to class.

Multiple objects can be created from the same class.

[B] Fill in the blanks:

The two major components of an object are_____ &_____.

The ability of a function or operator to act in different ways on different data types is called_____.

The process of building new classes from existing ones is known as_____.

If a class A inherits its properties from class B, then A and B are known as_____ class and_____ class, respectively.

Pascal and C are_____languages, whereas, C++ is_____ language.

[C] Match the following:

- | | | |
|----------------------------|------|----------------------------|
| (a) OO programming | (1) | Class |
| (b) Structured programming | (2) | Generic functions / class |
| (c) User-defined type | (3) | Base class, Derived class |
| (d) C, Pascal | (4) | Data and functions |
| (e) Object contents | (5) | Interaction of objects |
| (f) Inheritance | (6) | Structured prog. languages |
| (h) Templates | (7) | OO prog. languages |
| (i) C++, Java | (8) | Exceptions |
| (j) Containership | (9) | Interaction of functions |
| (k) int, float | (10) | User-defined types |
| (l) Employee, Student | (11) | Standard data types |
| (m) OO Error handling | (12) | Nested objects |

[D] Answer the following:

What is the basic difference between structured programming model and object oriented programming model?

What is the difference between a class and an object?

Give at least 5 examples of classes and objects.

What do you mean by encapsulation?

What do you mean by Inheritance?

What do you mean by polymorphism?

KanNotes

Structured programming encourages interaction of functions

OO programming encourages interaction of objects

A class is a user-defined type on the basis of which objects are created

An object contains specific data values and functions that can access or and/or manipulate them

Data members and member functions are encapsulated in an object

Data hiding means denying direct access to data from outside the object

Inheritance permits us to inherit properties of a base class into a derived class

Polymorphism permits us to use same function for different implementation in different derived classes

Containership lets us nest objects

Templates permit us to write generic functions/classes, leaving it for the compiler to create specific versions of them

C++ exceptions lets us handle run-time errors in OO manner

C++ promotes reuse of existing code using inheritance, containership and templates

Graduating to C++

C++ came into existence almost two decades after C. So C++ had the benefit of experience. This experience has made its way into improvements in some of the features that already existed in C language. It is important that we understand these features well, as we graduate from C to C++.

Comments

Input / Output in C++

Flexible Declarations

Flexible Initializations

Inferring Types

union and enum Syntax

Anonymous unions and enums

[Typecasting](#)

[void Pointers](#)

[The :: Operator](#)

[References](#)

[Types of Function Calls](#)

[Returning by Reference](#)

[The const Qualifier](#)

[const Pointers](#)

[const References](#)

[Returning const Values](#)

[const Member Functions](#)

[bool Data Type](#)

Exercise

KanNotes

Like C, C++ began its life at Bell Labs, where Bjarne Stroustrup developed the language in 1979. Stroustrup based C++ on C because of C's brevity and its widespread availability. In fact, every C program is also a valid C++ program. This means data types, control instructions, functions, arrays, strings, structures, unions, enums and pointers work exactly the same way in C++ as they did in C. Hence while migrating from C to C++ you do not have the burden of unlearning anything.

Since its initial version, C++ has undergone several significant revisions. The important milestones in this evolution are C++98, C++11, C++14 and C++17. All programs in this book work with C++17 compliant compilers like VisualStudio or gcc.

Main purpose of creating C++ was to make writing good programs easier for the programmer. To achieve this goal, Stroustrup added OOP features to C without significantly changing the C component. Thus, C++ is a superset of C. In order to support OOP features a number of new keywords were introduced in C++. [Figure 2-1](#) gives a list of these new keywords.

asm	bool	catch	class	const_cast
decltype	delete	dynamic_cast	explicit	false
friend	inline	mutable	namespace	new
operator	private	protected	public	reinterpret_cast
static_cast	template	this	throw	true
try	typeid	typename	using	virtual
wchar_t				

Figure 2-1. New keywords in C++.

We will postpone the discussion of these keywords and the object-oriented features of C++ to subsequent chapters. In this chapter we will take a look at the non-object-oriented extensions provided in C++. Let us begin with the comments are written in C++.

Comments

Unlike C, C++ supports two types of comments. C programmers are familiar with the `/* */` style of commenting. Anything lying within the `/*.. */` pair is ignored by the compiler. C++ additionally supports the `//` notation for commenting. For example,

```
c = 5.0 / 9 * (f - 32); // conversion formula
```

Here, everything following `//` till the end of line is treated as comment. Usually `/* */` style is used for commenting out a block of code, whereas, `//` is used for single line comments.

Input / Output in C++

Consider the following statement:

```
cout << "God, Give me common sense!" ;
```

This statement causes the phrase in the quotation marks to be displayed on the screen. A complete description of this statement requires an understanding of objects, operator overloading, and other topics we won't discuss until later in the book, but here's a brief preview.

The identifier **cout** (pronounced “C out”) is actually an object. It is predefined to correspond to the **standard output**. A **stream** is an abstraction that refers to a flow of data. The standard output stream normally flows to the screen display, although it can be redirected to other output devices. We will discuss streams in [Chapter](#).

The operator `<<` is called the **insertion** or **put to** operator. It directs the contents of the variable on its right to the object on its left. In our statement it directs the string constant “God, Give me common sense!” to which sends it to display.

If you know C, you will recognize `<<` as the left-shift bitwise operator and wonder how it can also be used to direct output. In C++ operators can be That is, they can perform different activities, depending on the context. We'll learn about overloading in [Chapter](#)

Although the concepts behind the use of `cout` and `<<` may be obscure at this point, using them is easy. They'll appear in almost every program in this book. Now that we know how output to the screen can be accomplished let us turn our attention to receiving input from the keyboard.

To receive input through the keyboard an identifier called `cin` is used. The word `cin` (pronounced “C in”) is an object, predefined in C++ to correspond to the **standard input**. This stream represents data coming from the keyboard (unless it has been redirected). The `>>` is the **extraction** or **get from** operator. It takes the value from the stream object on its left and places it in the variable on its right.

Here's a program that puts `cout` and `cin` to work.

```
#include  
using namespace std;
```

```
int main()
{
char str[40];
int m1, m2, m3, avg;

cout << "Enter your name:";
cin >> str;

cout << "Enter marks in three subjects:";
cin >> m1 >> m2 >> m3;
avg = (m1 + m2 + m3) / 3;

cout << "Your name is" << str;
cout << endl << "And your average marks are" << avg <<
endl;
return 0;
}
```

Here is some sample interaction with the program.

```
Enter your name: Saurav Gupta
Enter marks in three subjects: 44 72 64
Your name is Saurav Gupta
And your average marks are 60
```

In the first line of this program we have included the file ‘iostream’. Unlike C, in C++ header files do not have .h extension.

The ‘iostream’ file contains declarations that are needed by **cout** and **cin** identifiers, and << and >> operators. Without these declarations the compiler won't recognize **cout** and **cin** and will think that << and >> are being used incorrectly.

The next line, **using namespace** will be present in all the programs in this book. To understand the meaning of this statement we need to first understand what a namespace is. A namespace is nothing but a collection of identifiers (variable names and some other types of names that we haven't discussed yet) that all belong to a group or family. For example, all identifiers in the C++ standard library belong to a namespace called

There are two ways to refer to a specific identifier that belongs to a namespace. These are:

Use the **using** statement at the beginning of the program as we have done in our program.

Prefix the identifier with the name of the namespace followed by two colons. An example of this is shown below:

```
std::cout << "Enter your name:";
```

Naturally, the first way is a better way rather than preceding every **cin** and **endl** with

The first pair of *cout* and **cin** used in the program is straightforward. **cout** outputs a message asking for the name and **cin** receives that name in the string

Note the repeated use of the extraction operator **>>** in the second **cin** statement. This is perfectly legal. This is known as cascading of operators. It is a better idea than having three independent However, using this capability eliminates the opportunity to prompt the user between inputs. As you may have guessed, cascading of insertion operators is also allowed.

Look at the last **cout** statement. You would find the use of an unfamiliar word in this statement, This is known as a It causes a linefeed to be inserted in the output stream. As a result, the phrase following it appears on a fresh line. The same effect could have been obtained by sending the newline character '\n' to the output stream through any of the following statements:

```
cout << "\n" << "And your average marks are" << avg;  
cout << '\n' << "And your average marks are" << avg;  
cout << "\nAnd your average marks are" << avg;
```

These are workable statements. However, using **endl** to do the same job is perhaps a cleaner way of doing it.

And now a thousand-dollar question! Have we gained anything out of using **cin** and **cout** instead of **scanf()** and **Certainly**. In **scanf()** and **printf()** we have to always mention the format specifiers in the format string for integer, **%s** for string etc.). This gets completely avoided while using **cin** and **cout**. Also, as you will see in [Chapter](#) we can employ **cin** and **cout** to perform input/output for user-defined data types as well.

Flexible Declarations

C is quite rigid as far as declaration of variables is concerned. It requires that all variables be declared before the first executable statement. As against this, C++ allows definition of variables at the point where they are used. The following example illustrates this.

```
#include
using namespace std;
int main()
{
    int f;
    cin >> f;
    int c = (f - 32) * 5 / 9;
    cout << c;
    for (int j = 10; j <= 100; j++)
        cout << endl << j << endl;
    return 0;
}
```

Note that the variables **c** and **j** have not been declared at the beginning of the function. Instead, they have been declared where they are used to store a value. Declaring variables

where they are used makes the program easier to understand, since you don't need to refer repeatedly to the beginning of the function to find the variable declarations. However, this practice should be used with discretion. Variables that are used in many places in a function are probably better defined at the start of the function.

Flexible Initializations

C++ offers multiple ways to initialize a variable. The following code snippet shows this.

```
int age1 = 32;  
int age2 (32);  
int age3 {32};
```

All the three ways of initializing variables shown above are valid and equivalent.

All these statements will assign the value 32 to the 3 variables used.

Inferring Types

C++ compiler is also able to infer the type of a variable from the value being assigned to it. For this, we need to use **auto** as the type specifier for the variable as shown below.

```
auto age1 = 32;  
auto age2 = age1;
```

Since 32 is an type assumed for **age1** will be Similarly, since **age1** is **int** and it is being assigned to **age2** is also assumed to be an

Do we need to initialize a variable for the Compiler to infer the type of the variable correctly? No, it is not necessary to initialize the variable. We can do it as shown below.

```
char ch;  
decltype (ch) dh;
```

Here type of **dh** will be same as type of

auto and **decltype** are features that have been recently added to the language. We have used them in very simplistic examples. They are more meaningful when type cannot be obtained by other means.

union and enum Syntax

Consider the following declarations:

```
struct employee
{
    char name[20];
    int age;
};

union data
{
    char ch[2];
    int i;
};

enum status {married, unmarried, divorced};

employee e;
data d;
status s;
```

Note how the variables **d** and **s** have been defined. C++, unlike C, doesn't need the keywords **union** and **enum** while defining the variables **d** and **s**. Thus the format for defining a structure variable is exactly same as the one used for a variable of a built-in data type like **int** as shown below.

```
int i;  
employee e;
```

This similarity is not accidental. One of the aims of C++ is to ensure that your experience of working with user-defined data types is similar to that of built-in data types.

There is another major difference between structures and unions of C++ as compared to those in C. Structures and unions of C++ can contain as their elements not only various data types but also functions. We will study this aspect of structures and unions in [Chapter 5](#) when we study classes in C++.

Anonymous and

C++ permits us to use **anonymous union** and An anonymous union does not have a union name (tag), and its elements can be accessed directly without using a union variable.
Following example illustrates this:

```
union
{
    int i; char ch[2];
};
```

Both **i** and the array **ch[]** share the same memory locations and can be accessed directly simply by saying,

```
i = 10; ch[0] = 'A';
```

On similar lines anonymous can be built as shown below.

```
enum {sleeper, achtreetier, actwotier, acfirstclass};
int bogietype = actwotier;
```

Anonymous union is useful when it is nested inside a structure as shown below.

```
struct employee
{
    char name[20];
    union
    {
        int emptytype;
        char grade[4];
    };
};

int main()
{

    struct employee e;
    e.emptytype = 3;
    return 0;
}
```

In this declaration **emptytype** and **grade** share same memory locations and can be accessed directly. This direct access is possible because of anonymous union. Had the union been named, we would have been required to use multiple operators to reach **emptytype** or

The stream I/O classes presented in [Chapter 9](#) use several anonymous enumeration types.

Typecasting

If we carry out an operation between an **int** and a **float** the **int** is promoted to a **float** before performing the operation. This conversion takes place automatically. A few more such automatic conversions are possible in C/C++. As against these automatic conversions, to carry out data conversions desired by the programmer typecasting is used. Two different types of typecasting syntaxes are supported in C++. These are given below:

```
int y = 1001, j = 365, n;  
n = (y - 1) * (long) j; // C style typecasting, also supported by  
C++  
n = (y - 1) * long (j); // new C++ style typecasting
```

C supports only the first type of casting, whereas, C++ supports both. Instead of using typecasting we may as well have defined **j** as **l**. In this small fragment of code this would have worked. However, in a situation where we subsequently want to use **j** as an **int** there is no alternative to typecasting.

void Pointers

The keyword **void** can be used to define a pointer to a generic term. Unlike C, in C++ special care has to be taken to handle the assignment of **void** pointers to other pointer types. This is shown in the following code fragment.

```
void *p;  
char *s;  
p = s;  
s = p;
```

Here, the second assignment will flag an error indicating a type mismatch. While you can assign a pointer of any type to a **void** pointer, the reverse is not true unless you specifically typecast it as shown below:

```
s = (char *) p;
```

The :: Operator

:: is the scope resolution operator. It resolves a dispute in scope when local and global variables exist with same names. Let us straightway put it to work in a program.

```
#include
using namespace std;

int num = 10;
int main()
{
    int num = 15;
    cout << "Local num =" << num << endl;
    cout << "Global num =" << ::num << endl;
    ::num = 20;
    cout << "Local num =" << num << endl;
    cout << "Global num =" << ::num << endl;
    return 0;
}
```

Here is the output of the program...

```
Local num = 15
Global num = 10
Local num = 15
Global num = 20
```

Note that here we have two variables by the same name One is a global variable, and the other is a local variable. C++ allows you the flexibility of accessing both the variables. It achieves this through a scope resolution operator

Thus in the above program using **num** allows us an access to the local whereas, **::num** allows an access to the global The output proves that we can not only access a global variable and print its existing value, but also assign it a new value.

References

A reference is like a pointer. But, there is a subtle difference between the two. This difference we would see later. Let us first write a program that uses a reference.

```
#include
using namespace std;

int main()
{
    int i = 10;
    int &j = i;
    cout << "i =" << i << "j =" << j << endl;
    j = 20;
    cout << "i =" << i << "j =" << j << endl;
    i = 30;
    cout << "i =" << i << "j =" << j << endl;
    i++;
    cout << "i =" << i << "j =" << j << endl;
    j++;
    cout << "i =" << i << "j =" << j << endl;
    cout << "address of i =" << &i << "address of j =" << &j <<
    endl;
```

```
return o;  
}
```

And here is the output...

```
i = 10 j = 10  
i = 20 j = 20  
i = 30 j = 30  
i = 31 j = 31  
i = 32 j = 32
```

```
address of i = 61342 address of j = 61342
```

In this program **j** is called **reference** of A reference is indicated by using the **&** operator in the same way you use the ***** operator to indicate a pointer. A reference contains address of the variable that it is referring to. However, it is different than pointer in one way—to access the value that the reference is pointing to, we do not have to use the ***** operator. In other words, a reference gets automatically dereferenced.

So in our program once **j** starts referring to anytime we use in reality what gets used is Since **j** contains address of ***j** yields That is why when we attempt to assign a value 20 to it gets assigned to i.e. to Similarly, when we increment i.e. **i** gets incremented. Not only that, when we attempt to print the address of **j** what gets printed is address of i.e. address of

Cross check that with the output of the program shown above.

A few points to note about references...

A reference must always be initialized. Thus the following set of statements produce an error.

```
int i = 4;  
int &j; // error  
j = i;
```

Once a reference variable has been defined to refer to a particular variable, it cannot refer to any other variable. That is, once the variable and the reference are linked they are tied together inseparably.

The way we can create a reference to an **int** or a **float** or a **char** we can also create a reference to a pointer. The declaration of such a reference would look like this:

```
char *p = "Hello";  
char *&q = p;
```

A variable can have multiple references. Changing the value of one of them effects a change in all others.

Though an array of pointers is acceptable, an array of references is not.

TYPES OF FUNCTION CALLS

In C programming two types function calls were possible—call by value and call by address. In C++ there is one more type of function call feasible. It is known as call by reference. This call works the same way as call by address. That is, it both these calls we can change the values of the actual arguments through the formal arguments used in the called function. However, the code in a call by reference is much cleaner which makes it more readable, as the following program will justify.

```
#include  
using namespace std;  
  
// prototype declarations  
void swapv (int, int);  
void swapr (int &, int &);  
void swapa (int *, int *);
```

```
int main()
{
int a =10, b = 20;

swapv (a, b); // call by value
cout << endl << a << "\t" << b;
swapa (&a, &b); // call by address

cout << endl << a << "\t" << b;
swapr (a, b); // call by reference
cout << endl << a << "\t" << b << endl;
return 0;
}

void swapv (int i, int j)
{
int t;
t = i;
i = j;
j = t;
cout << i << "\t" << j;
}

void swapa (int *i, int *j)
{
int t;
t = *i;
*i = *j;
*j = t;
}
```

```
void swapr (int &i, int &j)
{
int t;
t = i;
i = j;
j = t;
}
```

In this program the call to **swapv()** is a call by value. Interchanging the values of **i** and **j** in this function has no effect on the values of **a** and **b** in **main()**. The **swapa()** function demonstrates a call by address, whereby using pointers we change the values of **a** and **b** in the calling function. The **swapr()** achieves the same purpose as the only difference being this function is called by reference.

You would agree that the style of using references to swap values is more elegant and pleasant to the eye. Moreover, a pointer has to be de-referenced before you can access a value using it, while a reference need not be. A reference works more directly in that sense. The calling syntax of **swapr()** too is much simpler as compared to that of

Thus **referencing** offers a clean, elegant and efficient way to pass parameters to functions that intend to change their

values. A call by reference is useful in one more situation. This is discussed below.

When a call by value is made a copy of the parameters being passed is made in the function being called. If the size of parameters is big this results in wastage of precious memory space. As against this, in a call by address or a call by reference a copy of the parameters being passed is not made. This is because in both calls addresses get passed. Given below is a program that shows the three types of calls.

```
#include
using namespace std;
struct employee
{
    char name[20];
    int age;
    float salary;

};

void displayVal (employee);
void displayAddr (employee *);
void displayRef (employee &);

int main()
{
```

```

employee e = {"Sanjay", 32, 3200.00};
displayVal (e);
displayAddr (&e);
displayRef (e);
return o;
}
void displayVal (employee e)
{
cout << e.name << endl << e.age << endl << e.salary << endl;
}
void displayAddr (employee *p)
{
cout << p->name << endl << p->age << endl << p->salary <<
endl;
}
void displayRef (employee &p)
{
cout << p.name << endl << p.age << endl << p.salary <<
endl;
}

```

Of the three function calls used in this program, the call by reference is best, as it avoids the copy as well as the usage of pointers.

RETURNING BY REFERENCE

We have seen how we can pass a reference to a function as a parameter. We can also return a reference from a function. When a function returns a reference, the function call can exist in any context where a reference can exist, including on the left hand side of an assignment. The following example would clarify this.

```
#include
using namespace std;

struct emp
{
    char name[20];
    int age;
    float sal;
};

emp e1 = {"Amol", 21, 2345.00};
emp e2 = {"Ajay", 23, 4500.75};
emp &fun();

int main()
{
    fun() = e2;
    cout << e1.name << endl << e1.age << endl << e1.sal << endl;
    return 0;
}
```

```
emp &fun()
{
cout << e1.name << endl << e1.age << endl << e1.sal << endl;
return e1;
}
```

Here we have declared a structure **emp** and initialized two global variables **e1** and **e2** to some values. In having printed the values stored in **e1** we have returned it by reference. What is strange is the call to the function It has been written on the left-hand side of the assignment operator:

```
fun() = e2;
```

fun() returns **e1** by reference. To this reference **e2** gets assigned. As the reference gets automatically dereferenced, **e2** get assigned to The following output of the **cout** statement in **main()** verifies that the assignment has indeed taken place.

```
Ajay
23
4500.75
```

A word of caution! Do not try to return a local variable by reference. This is because the local variable goes out of scope

when the function returns. You would, therefore, be returning a reference for a variable that no longer exists, and the calling function would be referring to a variable that does not exist. Some C++ compilers issue a warning when they see code that returns references to local variables. If you ignore the warning, you get unpredictable results. Sometimes the program appears to work because the stack location where the local variable existed is intact when the reference is used. A program that appears to work in some cases can fail in others due to device or multitasking interrupts that use the stack.

The **const** Qualifier

A variable can be marked as **const** (for constant), to specify that its value will not change throughout the program. Any attempt to alter the value of the variable defined with the **const** qualifier will result into an error message from the compiler. **const** is usually used to replace constants.

const qualifier ensures that your program does not inadvertently alter a variable that you intended to be a constant. It also reminds anybody reading the program that the variable is not intended to change. Variables with this qualifier are often named in uppercase, as a reminder that they are constants. The following program shows the usage of

```
#include  
using namespace std;  
  
int main()  
{  
    float r, a;  
    const float PI = 3.14f;  
    cin >> r;
```

```
a = PI * r * r;  
cout << endl << "Area of circle =" << a << endl;  
return 0;  
}
```

const is a better idea as compared to **#define** because its scope of operation can be controlled by placing it appropriately either inside a function or outside all functions. If a **const** is placed inside a function its effect will be localized to that function, whereas, if it is placed outside all functions then its effect will be global. We cannot exercise such finer control while using a **#define**.

CONST POINTERS

Look at the following program:

```
#include  
using namespace std;  
  
void xstrcpy (char *, char *);  
int main()  
{  
char str1[] = "Nagpur";  
char str2[10];
```

```
xstrcpy (str2, str1);
cout << str2 << endl;
return o;
}
void xstrcpy (char *t, char *s)
{
while (*s != '\0')
{
*t = *s;
t++;
s++;
}
*t = '\0';
}
```

This program simply copies the contents of **str1[]** into **str2[]** using the function. What would happen if we add the following lines beyond the last statement of

```
s = s - 6;
*s = 'K';
```

This would change the source string to “Kagpur”. Can we not ensure that the source string doesn't change even accidentally? We can, by changing the prototype of the function to

```
void xstrcpy (char *, const char *);
```

Correspondingly, the definition would change to:

```
void xstrcpy (char *t, const char *s)
{
// code
}
```

The following code fragment would help you to fix your ideas about **const** further.

```
char *p = "Hello"; // pointer is variable, string is constant
*p = 'M'; // error
p = "Bye"; // works
```

```
const char *q = "Hello"; // pointer is variable, string is
constant
*q = 'M'; // error
q = "Bye"; // works
```

```
char const *s = "Hello"; // pointer is variable, string is
constant
*s = 'M'; // error
s = "Bye"; // works
```

```
char * const t = "Hello"; // pointer is constant, so is string
*t = 'M'; // works
t = "Bye"; // error
```

```
const char * const u = "Hello"; // string is constant, so is
pointer
*u = 'M'; // error
u = "Bye"; // error
```

CONST REFERENCES

Earlier we saw that we can pass the reference of a variable to a function if we want to change the variable in the function without being required to use pointers. That is not the only use of passing by reference. If we pass the variable by value then it would be collected into another variable, thereby creating another variable. If several variables were passed by value, then those many additional variables would get created. We can avoid creation of these variables by passing the variables by reference. This would improve the efficiency. However, now there is a possibility that the variables may accidentally get modified in the function. This can be prevented by declaring them as constant. This is shown below.

```
#include
using namespace std;
```

```
void change (const int &);
```

```
int main()
{
int i = 32;
change (i);
cout << endl << i;
return 0;
}
void change (const int &j)
{
j = 45;
}
```

Passing **i** by reference prevents a new variable from getting created. At the same time using the **const** qualifier prevents **j** (and in turn from getting modified. Our attempt to modify **j** is met with an error message ‘Cannot modify a constant object’.

Let us now see a few examples that would further clarify the concept of the **const** qualifier and references.

```
#include
using namespace std;
```

```
int main()
{
const int i = 10;
int &j = i;
cout << "i =" << i << "j =" << j;
j = 20;
cout << "i =" << i << "j =" << j;
return 0;
}
```

Here we have tied **j** with **i**. So changing **j** should not change as we have declared **i** to be constant so it should not be possible to change it, even through its reference. So we should expect the following output from the program.

```
i = 10 j = 10
i = 10 j = 20
```

However, when we compile the program it reports an error saying ‘cannot convert from **const int** to **int**’. This error is reported because we have declared **j** to be a reference to an **int** and not to a **const**.

Let us take another similar case where we keep the reference constant.

```
#include
```

```
using namespace std;
```

```
int main()
{
int i = 10;
const int &j = i;
cout << i << j;
j = 20;
cout << i << j;
return 0;
}
```

This program won't even clear the compilation hurdle. Obviously so, since we are going back on our word. We declared that **j** would remain constant and then we are attempting to change it through the statement **j =**

Moral to be drawn from the above two programs is, once we declare a variable or a reference as constant we cannot change their values.

What do you think the following program would output?

```
#include
using namespace std;
```

```
int main()
{
int i = 10;
const int &j = i;
i = 20;
cout << endl << "i =" << i << "j =" << j << endl;
return 0;
}
```

The output would be:

```
i = 20 j = 20
```

What does that imply? The reference is dependent upon the variable it refers to. If we change the value of the variable, the value of the reference changes automatically, constant or otherwise.

The way we use referencing for integers can we do so with strings too? Of course! Here is how...

```
#include
using namespace std;
int main()
{
```

```

char *str1 = "Rain Rain Here Again";
char * &str2 = str1;
cout << endl << str1 << endl << str2;

*str1 = 'M';
cout << endl << str1 << endl << str2;

*str2 = 'P';
cout << endl << str1 << endl << str2;
return 0;
}

```

Here **str1** is a pointer to a constant string “Rain Rain Here Again”. And **str2** is a pointer that acts as reference to **str1**. Thus, both **str1** and **str2** point to the same string. Next we have attempted to change the constant string first through **str1** and through **str2**. Both attempts result into a runtime error, as through them we are trying to change a string which is constant.

RETURNING CONST VALUES

A function can return a pointer to a constant string as shown below.

```
#include
```

```
using namespace std;
```

```
const char *fun();
int main()
{
    const char *p;
    p = fun();
    *p = 'A'; // Error
    cout << p;
    return 0;
}
const char *fun()
{
    return "Rain";
}
```

Here since the function **fun()** is returning a constant string, we cannot use the pointer **p** to modify it. Not only this, the following operations too would be invalid:

main() cannot assign the return value to a pointer to a non-const string.

main() cannot pass the return value to a function that is expecting a pointer to a non-const string.

CONST MEMBER FUNCTIONS

In C structures can contain only data types, whereas, in C++ they can contain functions too. Also, similar to structure, there is one more entity in C++ called class. It too can contain data members and member functions. **const** can be used to qualify member functions in a structure or a class. This usage of **const** would be discussed in [Chapter](#)

bool Data Type

The data type **bool** gets its name from George Boole, a century English mathematician who invented the concept of using logical operators with true or false values. These values are often called boolean values.

This data type can take only two values **true** or **false**. It is most commonly used to hold the results of comparisons. For example,

```
bool x, y;  
int a = 10, b = 20, c = 30;  
x = a < b;  
y = b >= c;
```

Here **x** gets a value whereas, **y** gets value

By definition, **true** has a value **1** when converted to an integer and **false** has a value **0**. Conversely, integers can be implicitly converted to **bool** values—non-zero integers convert to **true** and **0** converts to **false**. They can be used in arithmetic and logical expressions. If so done, they are converted to integers and

integer arithmetic and logical operations are performed on these integers. The result of these operations can be converted back to a `o` is converted to `false` and a non-zero value is converted to `1`. The following program and its output would help you to understand these conversions.

```
#include
using namespace std;

int main()
{
    bool b = 32;
    int i = false;

    cout << endl << b << endl << i;

    int j = b + b;
    bool k = b + b;
    cout << endl << j << endl << k << endl;
    return 0;
}
```

Here is the output of the program...

```
1
0
```

2

1

bool can also be used as a return type of a function, usually indicating the success or failure of the function to carry out the assigned job. Older C++ compilers like Turbo C++ 3.0 doesn't recognize the **bool** data type.

Exercise

[A] State True or False:

In C++, a structure can contain data members, as well as functions that can operate upon the data members.

In C++, a union can contain data members, as well as functions that can operate upon the data members.

It is possible to create an array references.

Once a reference is tied with a variable it cannot be tied with another variable.

A variable can be tied with several references.

In C++ a function call can occur even on the left-hand side of an assignment operator.

It is unsafe to return a local variable by reference.

cin and **cout** are objects.

C++ permits the use of anonymous unions.

A pointer of another type can be assigned to a **void** pointer without the need for typecasting.

The following two definitions are same:

```
enum grade g;  
grade g;
```

The following two statements perform the same job:

```
int a = 10;  
int a (10);  
int a {10};
```

The following two statements perform the same job:

```
bool a;  
BOOL a;
```

The following three statements perform the same job:

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

```
cout << '\n';
cout << endl;
```

[B] What will be the output of the following programs:

(a) #include

```
using namespace std;
int main()
{
    int i = 5;
    int &j = i;
    int p = 10;
    j = p;
    cout << i << endl << j << endl;
    p = 20;
    cout << i << endl << j << endl;
    return 0;
}
```

(b) #include

```
using namespace std;
int main()
{
    char *p = "hello";
    char *q = p;
```

```
cout << p << endl << q << endl;
q = "Good Bye";
cout << p << endl << q << endl;
return o;
}
```

(c) #include
using namespace std;

```
int i = 20;
int main()
{
int i = 5;
cout << i << endl << ::i << endl;
return o;
}
```

(d) #include
using namespace std;
int i = 20;
int main()
{
int i = 5;
cout << i << endl << ::i << endl;
{
int i = 10;
cout << i << endl << ::i << endl;

```
}

return o;

}
```

```
(e) #include
using namespace std;
const int i = 10;
int main()
{
const int i = 20;
cout << i << endl << ::i << endl;
cout << &i << endl << &::i << endl;
return o;
```

```
}
```

```
(f) #include
using namespace std;
int main()
{
int i;
cout << sizeof (i) << endl << sizeof ('i') << endl;
return o;
}
```

[C] Point out the errors, if any, in the following programs.

```
(a) #include
using namespace std;
int main()
{
for (int i = 1; i <= 10; i++)
cout << i << endl;
cout << i;
return 0;
}
```

```
(b) #include
using namespace std;
int main()
{
int i = 5;
int &j = i;
int &k = j;
int &l = i;
cout << i << j << k << l;
return 0;
```

```
}
```

```
(c) #include
using namespace std;
int main()
{
```

```
int a = 10, b = 20;
long int c;
c = a * long int (b);
cout << c;
return o;
}
```

(d) #include
using namespace std;
const int i = 10;
int main()
{
const int i = 20;
cout << &i endl << &::i;
return o;
}

(e) #include
using namespace std;
int main()
{
char *p = "Hello";
p = "Hi";
*p = 'G';
cout << p;
return o;

}

(f) #include
using namespace std;
int main()
{
enum result {first, second, third};
result a = first;
int b = a;
result c = 1;
result d = result (1);
return 0;
}

(g) #include
using namespace std;
int a = 10;
int main()
{
int a = 20;
{
int a = 30;
cout << a << ::a << ::::a;
}
return 0;
}

```
(h) #include
using namespace std;
struct emp
{
    char name[20];
    int age;
    float sal;
};
emp e1 = {"Amol", 21, 2345.00};
emp e2 = {"Ajay", 19, 2300.00};
emp &fun();
int main()
{
    fun() = e2;
    cout << endl << e1.name << endl << e1.age << endl << e1.sal;
    return 0;
}
emp &fun()
{
    emp e3 = {"Aditya", 21, 3300.75};
    return e3;
}
```

```
(i) #include
using namespace std;
int main()
{
```

```
char t[] = "String functions are simple";
int l = strlen (t);
cout << l;
return o;
}
```

[D] Answer the following:

In the following program how would you define if the first **cout** is to output “Internet” twice, whereas, the second **cout** is to output “Intranet” twice.

```
#include
using namespace std;
int main()
{
    char *p = "Internet";
    cout << p << q << endl;
    q = "Intranet";
    cout << p << q << endl;
    return o;
}
```

If **employee** is a structure, **REGS** is a **union** and **maritalstatus** is an **enum** then does there exist any other way in which the following definitions can be made:

```
struct employee e;  
union REGS i;  
enum maritalstatus m;
```

Can the following statements be written in any other way:

```
employee *p;  
p = (employee *) malloc (sizeof (e));  
float q;  
int a, b;  
q = (float) a / b;
```

Create four integers, four pointers to these integers and four references to them. Store these pointers and references in two arrays and print out the values of four integers using these arrays.

Complete the following program by defining the function **swap()** and its prototype such that the output of the program is 20 10.

```
#include  
using namespace std;
```

```
void swapa (int &, int &);  
int main()  
{  
    int a = 10, b = 20;  
    swapa (a, b);  
    cout << a << b;  
    return 0;  
}  
void swapa (int &x, int &y)  
{  
    swapb (x, y);  
}
```

When should we make a call by reference?

Why is using **const** a better idea than an equivalent

What are the advantages of **cout** and **cin** over **printf()** and

Is this a valid comment:

```
// This is an /* invalid */ comment
```

What does the following prototype indicate:

```
const char *fun (char const*, const char *);
```


KanNotes

Every C program is a valid C++ program too.

Different versions of C++ are C++98, C++11, C++14 and C++17

C++ supports two types of comments. Usually // type of comment is used for single line and /* .. */ for multiple lines

cout is an object of ostream class. It is used for sending output to screen

endl is used send '\n' to the screen

<< is an insertion operator

cin is an object of istream class. It is used for receiving input from keyboard

>> is an extraction operator

<<, >> can be cascaded

`ostream` and `istream` classes are declared in `iostream` header file

`cout` and `cin` objects are defined in a namespace called `std`

To use `cout` and `cin`, `istream` file must be included and a `using namespace` statement should be used at the beginning of the program

If `using namespace` statement is not used then `cout` and `cin` must be prefixed with `std::`:

`cin` and `cout` are better than `printf()` and `scanf()` on two counts:

- 1) There is no need to remember and use the format specifiers
- 2) They can be used to input/output multiple values in an object

A variable can be initialized in 3 ways:

```
int i = 20;  
int j (20);  
int k {30};
```

auto keyword can be used to infer the type of a variable from the value being assigned to it

decltype keyword can be used to infer the type of a variable without assigning a value to it

While defining structure, union or enum variable there is no need to use the keywords struct, union or enum

Anonymous unions and enums permit direct access to their elements without being required to create union or enum variables

Anonymous unions are useful when they are nested in a structure

A variable can be defined just before the point of usage

If a variable is defined in a loop, then it dies when the control goes out of the loop

Both typecasting operations are valid:

```
n = (y - 1) * (long) j;  
n = (y - 1) * long (j);
```

Any type of pointer can be assigned to a void pointer

To assign a void pointer to any other type of pointer, it must be suitably typecasted

The scope resolution operator :: is used to access a global variable by the same name as a local variable

References are constant pointers that get automatically dereferenced

A reference can be tied with only one variable

A variable may have multiple references

A reference to a reference is not possible

An array of references is not allowed

Three types of function calls exist in C++:

Call by value - pass values of actual arguments

Call by address - pass addresses of actual arguments

Call by reference - pass addresses of actual arguments

In a call by reference changing formal arguments in the function does not change actual arguments

In call by address and call by reference, using formal arguments, actual arguments can be changed

In a function called by address, to reach the actual arguments one has to use pointers and the associated syntax

In a function called by reference, to reach the actual arguments one has to use references and the associated syntax (simpler)

Call by address and call by reference are also useful while sending big objects to a function as they do not make a copy of the object being passed

When a function returns a reference, the function call can exist in any context where a reference can exist, including on the left hand side of an assignment

When a variable is marked as const its value will not change throughout the program

const is better than a #define as scope of const can be either local or global

#define always has a global scope

const can also be used to keep pointers, references and strings constant

If a reference is declared as constant then using it the variable that the reference is tied with cannot be modified

A structure and a class can contain member functions. They also can be marked as const

A bool data type can take only two values - true or false

If a bool is used in arithmetic, then a true value is treated as 1 and false is treated as 0

If a integer is used in a boolean expression then a non-zero value is treated as true and a 0 is treated as false

Functions

Program development in C revolved around functions. C++ takes them to an entirely different level. Functions in C++ are more mature and capable. Apart from the normal functions, C++ also has instance, static, friend and virtual functions too. This chapter will help you understand and appreciate the power of functions in C++.

Strict Prototype Checking

Default Values for Function Arguments

Function Overloading

Difference in Return Type

Difference Through `typedef`

Difference Through `const`

Different Goals, Same Name

Operator Overloading

Operator Overloading FAQs

Inline Functions

Why Depend on Compiler?

What is the Guarantee?

When to Use Them?

New Return Type Syntax

virtual and *friend* Functions

Exercise

KanNotes

Functions was the basic building block around which C programs were written. C++ has taken them to an entirely new level. C++ functions are more powerful and versatile in several ways. These enhancements were invented to make C++ programs safer and **more** readable than their C equivalents.

Some of the features that have been added to C++ functions have relevance only when we use objected-oriented features of C++, whereas others apply to OO as well as non-OO programs. In this chapter we will restrict ourselves to those enhancements of C++ functions which apply to OO as well as non-OO programs. These include the following:

Strict prototype checking

Default values for function arguments

Function overloading

Operator overloading

Inline functions

Let us now understand these features one by one.

Strict Prototype Checking

A function prototype is a declaration that defines the name of the function, its parameters and its return type. Following are a few examples of function prototype declarations:

```
float square (float);  
char * strConvert (char *, int);  
double nthRoot (float, float);
```

Each of this declaration clearly specifies the function name, number, order and the type of arguments each function is going to receive during a call, and the type of the value that each function will return.

No C++ function can be called unless its prototype is available to the compiler. Compiler uses it to crosscheck whether the types of the arguments and the type of return value is same in the call and the definition. This is known as strong type checking; something which C lacks. If there is a mismatch C++ compiler points it out immediately.

Also, In C the type-checking was a bit relaxed. For example, you could pass a **double** and receive it in a In C++ this will be flagged as an error/warning.

Prototyping functions may mean a bit more work when you initially write a program, but prototypes can be invaluable tools in preventing hard-to-find errors. C++ was designed to prevent many of the problems caused by sloppy programmers passing the wrong data types to functions. If you really need to pass a **double** as a you can do so using a type cast. That way you explicitly (rather than accidentally) pass a value of one type as an argument to a parameter of another type.

Default Values for Function Arguments

Functions in C++ have an ability to assign default values for function. If we not pass values for such arguments then their default values are used. If we pass values for such arguments then the default values are overridden and the passed values are used. Let us understand this with an example program.

```
#include
using namespace std;

int sum (int n1, int n2, int n3 = 0, int n4 = 0);
int main()
{
    int s1, s2, s3;
    s1 = sum (10, 20);
    s2 = sum (10, 20, 30);
    s3 = sum (10, 20, 30, 40);
    cout << s1 << endl << s2 << endl << s3 << endl;
    return 0;
}
int sum (int n1, int n2, int n3, int n4)
{
    return (n1 + n2 + n3 + n4);
```

}

Note the prototype declaration of It indicates that o will be used as value of **n3** and **n4** if we do not pass them in a call. Thus, in the first call to **n3** and **n4** are taken as o. In the second call 30 is used for whereas **n4** is taken as o. In the last call 30 and 40 are used for **n3** and Thus, the default values for arguments are used if values are not passed in the call.

Let us now understand a few rules about default values for function arguments.

The default value for arguments are mentioned only in the function prototype and not in the function definition. The compiler uses the prototype information to build a call, not the function definition.

Default values can be assigned only for trailing arguments and not to leading arguments or arguments in the middle. Thus the following declaration is wrong.

```
int sum (int n1, int n2, int n3 = 0, int n4);
```

Default value for an argument can be a global constant, a global variable, or even a function call. For example, the function prototype given below is acceptable.

```
int myfunc (int flag = display());
```

In this case if **myfunc()** is called without an argument, the value returned by **display()** will be treated as default value of

Default values for function arguments are useful in 2 cases:

While making a function call if you don't want to take the trouble of writing arguments which almost always have the same value.

They are also useful in such cases where, after having written a program we decide to increase the capability of a function by adding another argument. Using default arguments means that the existing function calls can continue to use old number of arguments, while new function calls can use more.

Function Overloading

In C every function in a program has to have a unique name. At times this becomes annoying. For example, if we are to find absolute value of a numeric argument we have to define following 3 functions for each numeric data type:

```
int abs (int i);
long labs (long l);
double fabs (double d);
```

All these functions do the same thing, so it seems unnecessary to have three different function names. C++ overcomes this situation by allowing the programmer to create three different functions with the same name. This is called **function**. The following program illustrates this.

```
#include
using namespace std;

int abso (int);
long abso (long);
double abso (double);
```

```
int main()
{
int i = -25, j;
long l = -100000L, m;
double d = -12.34, e;

j = abso (i);
m = abso (l);
e = abso (d);

cout << j << endl << m << endl << e << endl;
return o;
}
```

```
int abso (int ii)
{
return (ii > o ? ii : ii * -1);
}
long abso (long ll)
{
return (ll > o ? ll : ll * -1);
}
double abso (double dd)
{
return (dd > o ? dd : dd * -1);
}
```

How does the C++ compiler know which of the **abso()**s should be called when a call is made? It decides that from the type of the argument being passed during the function call. For example, if an **int** is being passed the integer version of **abso()** gets called, if a **double** is being passed then the double version of **abso()** gets called and so on. That's quite logical, you would agree.

What if we make calls like,

```
char ch = abso ('A');  
float f = abso (3.14f);
```

We have not declared an **abso()** function to handle a **char** or a **float**. Hence C++ compiler calls the **int** version in the first case and the **double** version in the second. Had we defined these versions, then those versions would have been called.

Note that we haven't used the name **abs()** as it conflicts with the function by the same name present in the C++ standard library.

Let us now look at some of the finer issues related with function overloading.

DIFFERENCE IN RETURN TYPE

Consider the following program to convert a numeric string to a number.

```
#include
using namespace std;

int stringToNumber (char *);
long int stringToNumber (char *);

int main()
{
    int n1;
    long int n2;
    char *ptr1 = "155";
    char *ptr2 = "400000";

    n1 = stringToNumber (ptr1);
    n2 = stringToNumber (ptr2);
    cout << n1 << endl << n2;
    return 0;
}
```

Even if we define the overloaded functions `stringToNumber()` we will still get compilation errors. This is because these functions will receive the same type of argument and differ only in their return type. The overloaded functions must at least differ in the type, number or order of parameters they accept. Difference only in return type is not enough to consider them as overloaded functions.

DIFFERENCE THROUGH `TYPEDEF`

Can we fool the C++ compiler into believing that two data types are different by renaming one of them using `typedef` ? No. A `typedef` merely gives another name for an existing type and does not constitute a different type.

Hence, the following program segment will report an error.

```
typedef int INT;  
void display (int);  
void display (INT);
```

Here **INT** is just another name for an Hence compiler cannot differentiate between the two versions of

DIFFERENCE THROUGH `CONST`

Consider the following program to display two strings.

```
#include
using namespace std;

void display (char *);
void display (const char *);

int main()
{
    char *ch1 = "Hello";
    const char *ch2 = "Bye";
    display (ch1);
    display (ch2);
    return 0;
}

void display (char *p)
{
    cout << p << endl;
}

void display (const char *p)
{
    cout << p << endl;
}
```

This program has valid overloaded function **display()** since the types **char *** and **const char *** are different. First version of **display()** receives a pointer to a string whereas the second receives a pointer to a constant string.

DIFFERENT GOALS, SAME NAME

It's a bad programming idea to create overloaded functions that perform different types of actions; functions with the same name should have the same general purpose. For example, if we write an **abso()** function that returns the square root of a number, it would be both silly and confusing.

We must use overloaded functions judiciously. Their purpose is to provide a common name for several ‘similar but slightly divergent’ functions. Overusing overloaded functions feature unnecessarily can make a program unreadable.

Operator Overloading

Operator overloading is one of the most fascinating features of C++. It can transform complex, obscure program listings into intuitively obvious ones. Operators like +, *, -, <=, >=, etc. work only on standard data types like etc. However, in C++ we can overload these operators to make them work even with user-defined data types.

Suppose we wish to perform complex number arithmetic. As we know, there is no standard data type available for representing a complex number. So we can use a structure to declare a user-defined type for it as shown below.

```
struct Complex
{
    double real, imag;
};
```

Then we can create variables of this type and perform common operations like addition, subtraction and multiplication as shown in the following code snippet:

```
Complex a, b, c, d, e;  
c = add (a, b);  
d = subtract (a, b);  
e = multiply (a, b);
```

Instead of defining **subtract()** or **multiply()** functions the following form will be more intuitive:

```
Complex a, b, c, d, e;  
c = a + b;  
d = a - b;  
e = a * b;
```

But C++ doesn't know how to perform +, - or * on user-defined **Complex** types. Hence, we need to teach it by overloading these operators. How this can be done is shown in the following program:

```
#include  
using namespace std;  
  
struct Complex  
{  
    double real, imag;  
};
```

```
Complex setComplex (double r, double i);
void printComplex (Complex c);
Complex operator + (Complex c1, Complex c2);
Complex operator - (Complex c1, Complex c2);

int main()
{
Complex a, b, c, d;

a = setComplex (1.0, 1.0);
b = setComplex (2.0, 2.0);

c = a + b;
d = b + c - a;

cout << "c =";
printComplex (c);
cout << "d =";
printComplex (d);
return 0;
}

Complex setComplex (double r, double i)
{
```

```
Complex temp;
temp.real = r;
temp.imag = i;
return temp;
}

void printComplex (Complex t)
{
cout << "("<< t.real <<, "<< t.imag <<")"<< endl;
}

Complex operator + (Complex c1, Complex c2)
{
Complex temp;
temp.real = c1.real + c2.real;
temp.imag = c1.imag + c2.imag;
return temp;
}

Complex operator - (Complex c1, Complex c2)
{
Complex temp;
temp.real = c1.real - c2.real;
temp.imag = c1.imag - c2.imag;
return temp;
}
```

In this program the operator + and - are taught to operate on a user-defined data type. This is achieved by declaring a function using the keyword **operator** and the operator to be overloaded as shown below.

```
Complex operator + (Complex c1, Complex c2);
```

This declaration tells the compiler that the overloaded **operator + ()** function will receive two **Complex** types and return a **Complex** type. This function will be called if the addition operation is being done on operands of the type

In effect, operator overloading gives you the opportunity to extend the reach of C++ operators. Operator overloading really shines when we use it in conjunction with **classes** in C++. We will do this in [Chapter](#). Once we cover this I am sure you would be able to see the operator overloading concept in a different light.

OPERATOR OVERLOADING FAQS

Before we close this topic let me answer three most frequently asked questions about operator overloading.

Which operators cannot be overloaded?

The operators that cannot be overloaded are., ::, ? and ::.

If an operator has unary and binary forms (such as operators + or &), both can be overloaded.

What is the precedence of operator functions?

The operator functions have the same precedence as the intrinsic operations that use the same operator. For example, the * operator always has a higher priority over the + operator. There is no way to change operator precedence.

Is it possible to redefine intrinsic operators?

What you are really asking is, is it possible to create your own operator for adding a pair of The answer is no. Allowing you to change the behavior of intrinsic operations has the potential to make any program unreadable.

Inline Functions

One of the important advantages of using functions is that they help us save memory space. As all the calls to the function cause the same code to be executed; the function body need not be duplicated in memory.

Imagine a situation where a small function is being called several times in a program. There are certain overheads involved in every function call—time is spent in passing values, passing control, returning value and returning control. In C program these overheads can be avoided by defining an equivalent macro in place of a function. This improves speed of execution, but macros have some side effects, especially when they have arguments. For example, consider the following macro:

```
#define SQUARE(x) (x * x)
```

If we use it in the from

```
int y = SQUARE (3 + 1);
```

it will get expanded into $y = 3 + 1 * 3 + 1$ instead of $y = (3 + 1) * (3 + 1)$ as will be usually expected.

To avoid such undesirable side-effects, a concept of inline function has been introduced in C++.

When we use it, the function calls to it is replaced by the statements defined in the inline function. As there are no function calls now, the overhead of function calls gets avoided. The following program shows inline function at work.

```
#include
using namespace std;

inline void reportError (char *str)
{
    cout << endl << str;
    exit (1);
}

int main()
{
    // code to open source file
    if (fileOpeningFailed)
        reportError ("Unable to open source file");
```

```
// code to open target file
if (fileOpeningFailed)
reportError ("Unable to open target file");

// code to copy contents of source file into target file
return o;
}
```

Note that the function must be declared to be **inline** before calling it. On compilation the contents of the **reportError()** function would get inserted at two places within our program at places where **reportError()** is being called.

WHY DEPEND ON COMPILER?

One question that should occur to you is why ask the compiler to insert the code of the function in line with the other program code when we can easily do so ourselves? The trouble with repeatedly inserting the same code is that you lose the benefits of program organization and clarity that come with using functions. The program may run faster and take less space, but the listing is longer and more complex. Instead, if we write the code in an **inline** function the source file remains well organized and easy to read, since the

function is shown as a separate entity. However, when the program is compiled, the function body is actually inserted into the program wherever a function call occurs.

WHAT IS THE GUARANTEE?

Note that when we define the function **inline** there is no guarantee that its code would get inserted at the place where the call is being made. This is because we are just making a request to the compiler. The C++ language does not define under what conditions the compiler may choose to ignore our request. Because of this ambiguity in the language specification, compiler builders have flexibility in how they interpret the requirements.

WHEN TO USE THEM?

You should use the **inline** function qualifier only when the function code is small. If the functions are large you should prefer the normal functions since the savings in memory space is worth the comparatively small sacrifice in execution speed.

New Return Type Syntax

Since C++11 a new syntax called trailing return syntax has been introduced to indicate the type of value the function is going to return. For example, consider the following prototype declaration:

```
int addNum (int x, int y);
```

This declaration can also be written as

```
auto addNum (int x, int y) -> int;
```

Here the keyword **auto** does not perform any type inference. It is just part of the syntax for trailing return types.

In C++14, the **auto** keyword can also infer the function's return type. For example, in the following function since $x + y$ evaluates to an integer, the compiler infers that the return type of this function is

```
auto addNum (int x, int y)
```

```
{  
return x + y;  
}
```

In my opinion this syntax should be avoided, since in absence of a specific return type, the caller may misinterpret the return type which may lead to inadvertent errors.

Instance, virtual and friend Functions

In C++ we can have four more types of functions: **virtual** and **friend**. However all of these are related with classes. Hence, their discussion cannot be taken up until you know classes. instance function have been explained in [Chapter](#) **static** functions in [Chapter](#) **virtual** functions in [Chapter 8](#) and **friend** functions in [Chapter](#)

With this much knowledge under our belt we have achieved enough competence to launch ourselves into the object-oriented world... which we would do in the next chapter.

Exercise

[A] State True or False:

It is mandatory to declare prototype of a function being called.

Two functions can be overloaded if their arguments are similar but their return values are different.

Two functions can be overloaded only if their arguments differ in number, order or type.

If default values are mentioned for the four arguments in the function prototype, we can call this function and pass it the first and the fourth argument.

A function can be overloaded any number of times.

The assignment operator cannot be overloaded.

When we define the function to be **inline** there is no guarantee that its code will get inserted at the place where the call is being made.

The side effects of the macro definition get eliminated if we use **inline** functions.

[B] Point out the errors, if any, in the following programs.

(a) int main()
{
int a = 30;
fun();
return 0;
}
void fun()
{
int b = 20;
}

(b) #include
using namespace std;
void fun()
{
cout << "Hello";
}

```
int main()
{
fun();
return o;
}
```

(c) #include

```
using namespace std;
int fun (int, int);
int fun (int, int);
int main()
{
int a;
a = fun (10, 30);
cout << a;
return o;
}
void fun (int x, int y)
{
return x + y;
}
```

(d) #include

```
using namespace std;
int main()
{
```

```
void fun1 (void);
void fun2 (void);
fun1();
return o;
}
void fun1 (void)
{
fun2();
cout << endl << "Hi...Hello";
}
void fun2 (void)
{
cout << endl << "to you";
}
```

(e) #include
using namespace std;
void fun (int, float);
int main()
{
fun();
return o;
}
void fun (int i = 10, float a = 3.14)
{
cout << i << a;
}

```
(f) #include
using namespace std;

void fun (int = 10, int = 20, int = 30);
void fun (int, int);
int main()
{
    fun (1, 2);
    return 0;
}
void fun (int x, int y, int z)
{
    cout << endl << x << endl << y << endl << z;
}
void fun (int x, int y)
{
    cout << endl << x << endl << y;
}
```

[C] Answer the following:

Write a program which calls a function called This function should be capable of clearing a part of the screen. If the function is called without any arguments then it should clear the entire screen. Assume the maximum screen size to be of 25 rows x 80 columns.

Suppose there is a function with the following prototype:

```
void fun (int = 10, int = 20, int = 30, int = 40);
```

If this function is called by passing 2 arguments to it, how can we make sure that these arguments are treated as first and third, whereas, the second and the fourth are taken as defaults.

Write a program that defines a function **display()** whose prototype is given below.

```
void display (char ch = '*', int num = 80);
```

Call this function to display the lines shown below.

=====

Write overloaded functions to convert an **int** to a string and to convert a **float** to a string.

Write overloaded functions to convert a string to an **int** and to convert a string to a

Declare a structure called **matrix** containing a 3×3 array of integers. Overload the + operator to carry out addition of two matrices.

Write function prototypes for the following:

- A function which receives an **int** and a **float** and returns a double.
- A function that receives an **int** pointer and **float** reference and returns an **int** pointer.
- A function which doesn't receive anything and doesn't return anything.
- A function that receives an array of and a **float** reference and doesn't return anything.

Which operators cannot be overloaded?

Can we change the hierarchy of operators by overloading them?

KanNotes

It is necessary to declare the prototype of the function being called

C++ compiler uses strict prototype checking to pass a function call as valid

Default values can be assigned only for trailing arguments of a function in its prototype declaration

Multiple functions carrying similar jobs but differing in arguments can be overloaded

Overloaded functions have same name, but their arguments differ in number, order or type

Difference in return types of functions is not a sufficient criterion for functions to be overloaded

Difference in arguments in overloaded functions cannot be on the basis of typedefed names

Operators can be overloaded to make operations on user-defined types more intuitive

Precedence of operators cannot be changed using operator overloading

Operators., :: and ?: cannot be overloaded

Operator overloading cannot be done for standard types like int or float

Inline functions combine the advantage of function as well as a macro

A call to inline function is substituted by the code present in its definition

By marking a function as inline doesn't guarantee that C++ compiler will actually carry out inlining

Classes and Objects

Classes and objects are to OOP, what functions were to Structured programming. A good foundation about classes and objects would make you a sound C++ programmer.

Structures and Classes

Classes and Constructors

Destructors

A Complex Class

The *this* Pointer

Overloading Unary Operators

Objects and Memory

Structures and Classes Revisited

Ideal Program Organization

Exercise

KanNotes

Having familiarized ourselves with the non-object oriented extensions of C++ like default values for function arguments, overloaded functions, overloaded operators, call by reference, the **const** qualifier etc., it's time to move on the object oriented features of C++. Let us begin with structures and classes.

Structures and Classes

A structure in C is a collection of similar or dissimilar data types. C++ extends the reach of structures by allowing the inclusion of functions within structures. Placing data and functions (that work upon the data) together into a single entity is the central idea in object oriented programming.

There is another entity in C++ called **class** that too can hold data and functions. There is almost no difference in the syntax of a structure and a class, hence at least in principle they can be used interchangeably. However, most C++ programmers use structures to hold data and classes to hold both data and functions.

Let us begin with a program that demonstrates the syntax and general features of a class. Here's the listing of the program.

```
#include  
using namespace std;  
  
class Rectangle  
{
```

```
private:  
int len, br;  
public:  
void getData()  
{  
cout << endl << "Enter length and breadth";  
cin >> len >> br;  
}  
void setData (int l, int b)  
{  
len = l;  
  
br = b;  
}  
void displayData()  
{  
cout << endl << "length =" << len;  
cout << endl << "breadth =" << br;  
}  
void areaPeri ()  
{  
int a, p;  
a = len * br;  
p = 2 * (len + br);  
cout << endl << "area =" << a;  
cout << endl << "perimeter =" << p << endl;  
}  
};
```

```

int main()
{
    Rectangle r1, r2, r3; // define three objects of class Rectangle

    r1.setData (10, 20); // set data in elements of the object
    r1.displayData(); // display the data set by setData()
    r1.areaPeri(); // calculate and print area and perimeter

    r2.setData (5, 8);
    r2.displayData();
    r2.areaPeri();

    r3.getData(); // receive data from keyboard
    r3.displayData();
    r3.areaPeri();

    return 0;
}

```

Look at the **Rectangle** class declaration in our program. The word **class** is a keyword and is followed by which is the class name. Like a structure, the body of a class is delimited by braces and terminated by a semicolon. Variables declared within the class are called its **data members** and functions

declared in it are called **member functions** or The **Rectangle** class contains two data members— **len** and **br** and four member functions— **getData()**, **displayData()** and

The class contains two unfamiliar keywords— **private** and **public**. They control the access to the class members. **private** members of a class are not accessible from outside the class, whereas, **public** members are.

Though not a rule, usually data members are marked **private** and member functions are marked As a result, data members remain safe. They cannot be accessed or manipulated from outside the class. If we wish to access or manipulate them, then we must approach them through the **public** member functions of the class.

Within a class any member can access any other member. This concept of keeping data members safe and providing them access through member functions is called **data**. Since member functions provide systematic access to hidden data members, they are often called the **class**.

Don't confuse data hiding with the security techniques used to protect computer data. Security techniques prevent illegal users from accessing data. Data hiding, on the other hand, is used to protect well-intentioned users from honest mistakes.

Once the class is declared, let us see how **main()** makes use of it. To do this we need to create objects of this class. In **main()** we have created 3 objects **r2** and **r3** of the type

To draw a parallel, look at the following declarations:

```
int i;  
Rectangle r1;
```

Here, **int** is a standard data type, whereas **Rectangle** is a user-defined data type. **i** and **r1** are variables of the type **int** and **Rectangle** respectively. In OOP terminology **r1** is called object of the **Rectangle** class.

An object is often also termed as an **instance** of a class, and the process of creating an object is called **In**. In our program we have defined three objects **r2** and **r3**. Each object has its own **len** and **br** data members. These objects are shown in [Figure](#)

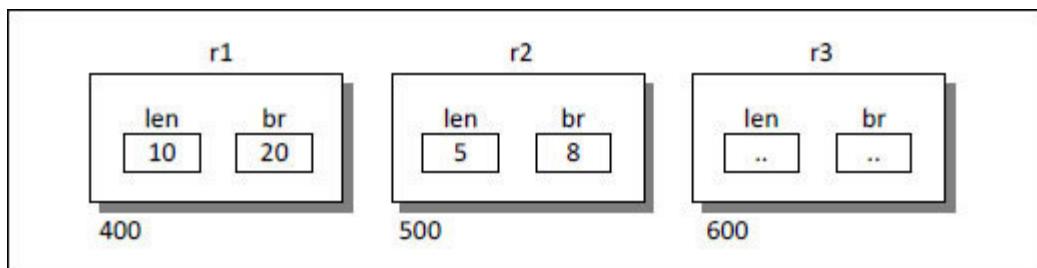


Figure 4.1

The next statement

```
r1.setData (10, 20);
```

calls the member function Since **setData()** is a member function of the **Rectangle** class, it must always be called using an object of this class. This is because a member function is always called to act on a specific object.

The dot operator connects the object name and the member function. The syntax is similar to the way we refer structure elements, but the parentheses signal that we are accessing a member function rather than a data member. The dot operator is also called ‘class member access operator’.

The first call to

```
r1.setData (10, 20);
```

executes the **setData()** member function of the **r1** object. This function sets the variables **len** and **br** of object **r1** to values **10** and **20** respectively. Likewise, the second call to **setData()** sets the values for variables in the second object.

The **displayData()** function displays the values of **len** and **br** of the object using which it is called. Likewise, **areaPeri()** calculates and prints the area and perimeter using **len** and **br** of the object using which it is called.

If we attempt a call

```
setData (10, 20);
```

compiler will report an error since we have not specified for which object are we planning to set the data.

Classes and Constructors

In the last section we had our first tryst with a class. Just to reiterate, a class contains data members and member functions. Let us now move one step further. Observe the following program carefully.

```
#include
using namespace std;

class Integer
{
private:
int i;
public:
void getData()
{
cout << endl << "Enter any integer";
cin >> i;
}
void setData (int j)
{
i = j;
}
```

```
Integer() // zero argument constructor
{
}
Integer (int j) // one argument constructor
{
    i = j;
}
void displayData()

{
    cout << endl << "value of i =" << i << endl;
}
};

int main()
{
    Integer i1 (100), i2, i3;

    i1.displayData();
    i2.setData (200);
    i2.displayData();
    i3.getData();
    i3.displayData();
    return 0;
}
```

This program shows three ways in which we can assign values to data members of an object. One is through the member function **setData()** to whom we pass the value to be set up. Another way is by receiving values through keyboard as shown in function `SetData()`. This brings us to the third method, which uses an entity called `Constructor`. It is a special member function that allows us to set up values while defining the object, without the need to make a separate call to a member function. Thus, constructor is a member function that is executed automatically whenever an object is created.

There are some unusual aspects to constructor functions. First, it is no accident that they have exactly the same name as the class of which they are members. In fact it's a rule that the class and the constructor function within it must have same names. This is how the compiler knows that the member function is a constructor.

Secondly, no return type is used for constructors. Why not? Since the constructor is called automatically when an object is created, returning a value would not make sense.

In our program the statement

```
Integer i1 (100), i2, i3;
```

creates three objects of the type As each is created, its constructor, is executed. So the effect of this single statement is to not only create three objects but also to initialize their **i** variables.

Note carefully that the **Integer** class has two constructors with the same name They are called **overloaded** Which of the two constructors gets called when an object is created depends on how many arguments are used in the definition of the object.

```
integer i2; // calls zero-argument constructor integer  
i1 (100); // calls one-argument constructor
```

A class may contain multiple overloaded constructors depending upon how we wish to initialize its objects.

Can we create an object from a class that doesn't have a constructor? Yes, because when we don't define a constructor the compiler inserts a zero-argument constructor in it. The zero-argument constructor is often called an **implicit** constructor or **default** constructor.

Note that if we define a one-argument constructor, we cannot rely on compiler to provide the default constructor. If we need it, we need to define it ourselves.

Finally, a constructor can be called for any object only once—when an object is created. If we wish to later on change the values in an object, we can do so by calling the **setData()** function on it.

Constructors

We've seen that a special member function—the constructor—is called automatically when an object is first created. Similarly, when an object is destroyed a function called destructor automatically gets called. A destructor has the same name as the constructor (which is same as the class name) but is preceded by a tilde (~). The following program shows destructor at work.

```
#include
using namespace std;

class example
{
private:
int data;
public:
example() // constructor (same name as class)
{
cout << endl << "Inside the constructor";
}
~example() // destructor (same name with tilde)
{
```

```
cout << endl << "Inside the destructor" << endl;
}
};

int main()
{
example e;
return 0;
}
```

When the object **e** gets created the constructor gets called.
When control goes outside **main()** the object **e** gets destroyed.
This invokes the destructor function. Like constructors,
destructors do not have a return value. They also take no
arguments (the assumption being that there's only one way to
destroy an object).

The most common use of destructors is to deallocate memory
that was allocated for the object by the constructor. We'll
investigate this issue further in a later section of this chapter.

A Complex Class

As we know, a complex number consists of a real part and an imaginary part. The following program puts the concept of overloaded constructors, operator overloading and destructor to a practical stint by developing a class to implement complex numbers.

```
#include
using namespace std;

class Complex
{
private:
    float real, imag;

public:
    Complex()
    {
    }
    Complex (float r, float i)
    {
        real = r;
```

```
    imag = i;
}
void displayData()
{
    cout << "real =" << real << "imaginary=" << imag << endl;
}
void addComplex (Complex x, Complex y)
{
    real = x.real + y.real;
    imag = x.imag + y.imag;

}

Complex addComplex (Complex y)
{
    Complex t;
    t.real = real + y.real;
    t.imag = imag + y.imag;
    return (t);
}

Complex operator + (Complex y)
{
    Complex t;
    t.real = real + y.real;
    t.imag = imag + y.imag;
    return (t);
}
};
```

```
int main()
{
Complex c1 (1.1f, 2.2f), c2 (3.5f, 4.4f), c3, c4, c5;

c3.addComplex (c1, c2);
c3.displayData();

c4 = c1.addComplex (c2);
c4.displayData();

c5 = c1 + c2;
c5.displayData();

return 0;
}
```

In this program we have used the 2-argument constructors to initialize objects **c1** and **c2**. Then we have used 3 different ways to add two complex numbers. These are shown below.

```
c3.addComplex (c1, c2);
c4 = c1.addComplex (c2);
c5 = c1 + c2;
```

As you can see, **addComplex()** is an overloaded function. In its first version, the two complex numbers to be added, i.e. **c1** and **c2** are passed to it. They are collected in suitably added, and stored in How come result of addition is stored in That is because we called the member function using

The second call to **addComplex()** has been done using In this call **c2** is passed to the function. Hence while performing addition **real** refers to real and **y.real** refers to The result of addition is stored in another **Complex** object At the end of addition **t** is returned and assigned to **c4** in

Let us examine the call to the overloaded ‘+’ operator more closely.

```
c5 = c1 + c2; // call
```

Compiler converts this expression into

```
c5 = c1.operator + (c2);
```

Thus call to the **operator +()** function is made using In this call **c2** is passed to the function. Hence while performing addition **real** refers to real and **y.real** refers to The result of addition is stored in another **Complex** object At the end of addition **t** is returned and assigned to **c5** in

As an exercise you may try to implement an overloaded operator * function to perform multiplication of two complex numbers. Also verify whether you get correct result for the operation

```
c6 = c1 + c2 * c3;
```

The this Pointer

We know that each object created from a class has its own set of data members. When we call member function using an object, the data members used in the member function refers to that object's data using which the member function is being called. Let us understand this with the help of a program.

```
class Sample
{
private:
int i;
float j;
public:
void setData (int x, float y)
{
i = x;
j = y;
}
int main()
{
Sample s1, s2;
```

```
s1.setData (10, 3.14f);
s2.setData (20, 6.28f);
}
```

Here the two **Sample** objects **s1** and **s2** have their own set of **i** and **j**. In **setData()** when we use **i** and **j** whose **i** and **j** are they, or For the ease of understanding we say that, if we make the call using **s1** then they are **i** and **j** and if we make a call using **s2** then they are **i** and **j**. The real mechanism that is at work is a special pointer called **this** pointer. Let us try to understand the working of **this** pointer.

When we make the call

```
s1.setData (10, 3.14f);
```

in addition to 10 and 3.14f, address of **s1** is also passed to This address is collected by **setData()** in a pointer called The type of **this** pointer is **Sample** When we use **i** and **j** in **setData()** they are used in the form

```
this->i = x;
this->j = y;
```

Since **this** contains address of **s1** for this call, **i** and **j** refer to **i** and **j**. When we make the call using **this** pointer contains address of hence **i** and **j** refer to **i** and **j**.

You can verify that this pointer contains addresses of **s1** and **s2** respectively for the two call by adding the following statement in

```
cout << "Current address in this =" << this << endl;
```

Note that **this** pointer is of the type **Sample ***. Hence it cannot change during the call and point to any other object other than the one that is used to call. Also, like any local variable, it dies when control returns from **setData()** function.

The **this** pointer is at work for every single call that is made using an object.

Overloading Unary Operators

As you know, a unary operators acts on only one operand. Examples of unary operators are the increment and decrement operators ++ and --, and the unary minus, as in -45.

Let us now implement an overloaded unary operator ++ for a class called While overloading this operator we must be able to distinguish between pre-increment and post-increment operations shown below.

```
d = ++c; // pre-increment  
f = e++; // post-increment
```

In the first case firstly **c** is incremented and then it is assigned to **d** against this, in the second case firstly **e** is assigned to **f** and then **e** is incremented. To distinguish between the two versions of overloaded ++ operator function the post-increment version uses int as argument. In true sense, the **int** isn't really an argument, and it doesn't mean integer. It's simply a signal to the compiler that the two versions of operator functions are different.

Given below is the implementation of **Index** class containing both versions of overloaded operator `++` function.

```
#include
using namespace std;
class Index
{
private:
int count;
public:
Index()
{
    count = 0;
}
Index (int i)
{
    count = i;
}
Index operator ++ ()
{
    ++count;
    return *this;
}
Index operator ++ (int)
{
    Index temp (count); // calls 1-argument constructor
    count++;
}
```

```
return temp;
}
void showdata()
{
cout << count << endl;
}
};

int main()
{
Index c, d, e, f;
cout << "c =";
c.showdata();
d = ++c;
cout << "c =";

c.showdata();
cout << "d =";
d.showdata();

cout << "e =";
e.showdata();
f = e++;
cout << "e =";
e.showdata();
cout << "f=";
f.showdata();
```

```
    return o;  
}
```

And here is the output of the program...

```
c = 0  
c = 1  
d = 1  
e = 0  
e = 1  
f = 0
```

Note the following points about the program:

The calls to overloaded operator function get converted into the form

```
d = c.operator ++ (); // calls pre-increment version
```

```
f = e.operator ++ (o); // calls post-increment version
```

Since the number of arguments is different for each call, it now becomes easy for the compiler to decide which call should be bound to which version of the overloaded function.

In both versions we are returning an Index object since it is to be assigned to **d** or

When the pre-increment version is called, **this** pointer contains address of object We increment **count** and return the object **c** through the expression ***this**. Instead of this we could have done it as shown below.

```
Index temp;  
++count;  
temp.count = count;  
return temp;
```

But creation of a new **temp** object is an overhead. It is better that we avoid it.

Objects and Memory

If an object is similar to a variable can we find out the number of bytes occupied by it in memory. Yes, using the **sizeof** operator that we use for standard data types. The following program illustrates this.

```
#include
using namespace std;

class Sample
{
private:
int i;
float a;
char ch;
public:
Sample (int j, float b, char dh)
{
i = j; a = b; ch = dh;
}
};
int main()
{
```

```
Sample s1 (10, 3.14f, 'A');  
Sample s2 (20, 6.28f, 'B');  
cout << sizeof (s1) << endl;  
cout << sizeof (s2) << endl;  
return 0;  
}
```

The program reports the size of the objects **s1** and **s2** as 12 bytes, which is the sum of sizes of **a** and **ch**. This means the size of any object is sum of sizes of its data members. This leads to an important question—do functions have no influence on the size of an object? The answer is no, since all the objects in a given class use the same member functions.

This makes sense as the functions for each object are identical. There's no point in duplicating all the member functions every time you create another object from a class. However, the data members will hold different values, so there must be a separate set of data members for each object.

In our program there are two objects of type **so** so there are two instances of **a** and **ch** in memory. However, there is only one copy of the constructor and the **display()** function. All the objects of the class share these functions. This does not lead to conflict of any type since only one function is executed at a time.

While programming rarely are you required to bother about whether there is a separate copy of a member function for each object or all objects are sharing a single copy. However, knowing what is happening behind the scenes has never harmed anybody.

Structures and Classes Revisited

In principle, we can use a structure at every place where a class is used. Thus, the two declarations given below are equivalent.

```
class Sample
```

```
{
```

```
    private :
```

```
        int data ;
```

```
    public :
```

```
        void fun( )
```

```
        {
```

```
            // some code
```

```
}
```

```
};
```

```
struct Sample
```

```
{
```

```
    private :
```

```
        int data ;
```

```
    public :
```

```
        void fun( )
```

```
        {
```

```
            // some code
```

```
}
```

```
};
```

The only formal difference between a class and a structure is that class members are **private** by default, while in a structure they are **public** by default. So, you may drop the keyword **private** from the above class declaration. However, in the interest of clarity, you are advised to use the keyword **private** explicitly.

Though in a structure all members are by default **public** we can't afford to drop the keyword. On doing so the **private**

clause will become applicable even to the function If we are still keen on dropping the keyword **public** we will have to define the function before the data members as shown below:

```
struct Sample
{
void fun()
{
// some code
}
private:

int data;
};
```

Though in principle we can use a structure at every place where a class is used, in most situations C++ programmers prefer to use structures to group data, and classes to group both data and functions.

Ideal Program Organization

In all programs in this chapter we have defined class and the client code that uses the class, i.e. **main()** in a single file. This is not an ideal way of organizing the program for two reasons. These are listed below.

If some other program needs the same class we need to extract the class from one file and copy it into the new program file. This becomes tedious if there are several classes in our program.

If some other programmer wants to use our class we have to part with the source code of the class. This is like giving away our trade secret.

To overcome these limitations we should ideally organize the program in 3 different files—a ‘.h’ file in which we merely declare the class, a ‘.cpp’ file in which we define the member functions of the class and a ‘.cpp’ file which uses the first two class files (often called the driver file or a client).

The class declaration in ‘.h’ file contains the data members and prototype declaration of member functions. The real meat of the class, i.e. definition of member functions remains in a separate ‘.cpp’ file. Thus this approach helps us to separate the interface from the implementation. This is considered to be a good software development practice.

Moreover, to other programmers who wish to use our class we can give only the compiled code of the class implementation and the ‘.h’ file. As we are not giving the ‘.cpp’ file, the implementation secret remains with us. We have to give the ‘.h’ file, otherwise, those who wish to use our class would never know the details like name of the class, names of its member functions, their parameters and return types, etc. In absence of this information they will never be able to use our class.

Also, this organization promotes reusability as now it is easily possible to copy the class files into another project which wishes to use the class.

Given below is the listing of a program that organizes the code in 3 files on the lines discussed above.

```
// Employee.h - Class declaration  
// This file declares the Employee class without revealing the
```

```
// implementations of its member functions #include
using namespace std;
class Employee
{
private:
string name;
int age;
public:
Employee();
Employee (string n, int a);
void getData();
void showData();
};
```

```
// Employee.cpp - Class implementation
// This file contains implementations of the member functions
// of
// Employee class
```

```
#include
#include "Employee.h"
using namespace std;
```

```
Employee::Employee()
{
name = "";
```

```
age = 0;
}
Employee::Employee (string n, int a)
{
name = n;
age = a;
}
void Employee::getData()
{
cout << endl << "Enter name and age" << endl;
cin >> name >> age;
}
void Employee::showData()
{
cout << "name =" << name << endl << "age =" << age <<
endl;
}
```

```
// Main.cpp - Client code
// This file contains code to use the Employee class whose
interface has
```

```
// been separated from its implementation
#include
```

```
#include "Employee.h"
using namespace std;
```

```
int main()
{
    Employee e1 ("Sanjay", 34);
    e1.showData();
    Employee e2;
    e2.getData();
    e2.showData();
    return 0;
}
```

There are a few important observations that you can make from the above program listings. These are given below.

Instead of a character array we have used a standard library class **string** for storing name of an employee. The advantage of using a **string** class over a character array is, it contains several functions that make string operations very easy.

The **string** class is defined in a header file ‘string’. So to use the class we need to include this file.

In the header file ‘Employee.h’ we have used **using namespace std** so that we are not required to use **std::string** wherever **string** is to be used.

The ‘Employee.cpp’ file contains definitions of Employee class's member functions. To make the Employee class available in this file we need to include ‘Employee.h’

We have included ‘Employee.h’ using “” rather than the usual < >. This is because ‘Employee.h’ will be in our project folder and not in the standard include path.

Each member function definition is preceded by the class name, and the scope resolution operator. This indicates that these functions belong to a class called **Employee** and are not global functions.

In **getData()** function we have received the name using **cin >>** This is ok so long as the name is a single word. If name is multiword, we can use the statement

```
getLine (cin, name);
```

In ‘Main.cpp’ it is necessary to include ‘Employee.h’. Without it we cannot use the **Employee** class in

Finally, how do we organize the code in three files? If you are using Visual Studio then follow the steps given below.

Create a project called ‘Client’. Type the code given above in ‘Client.cpp’.

Right click on project ‘Client’ in Solution Explorer. Select ‘Add | Class...’ from the menu that pops up.

Select ‘C++ class...’. Select ‘Add’ button. A dialog appears. Type class name as ‘Employee’, click on ‘Finish’ button. Two files ‘Employee.h’ and ‘Employee.cpp’ will get created.

Type the class declaration code in ‘Employee.h’ and class implementation code in ‘Employee.cpp’.

Build (compile and link) and execute the program by using Ctrl F5.

For other development environments similar steps available in that environment should be followed.

[Figure 4-2](#) shows how the compilation and linking will proceed.

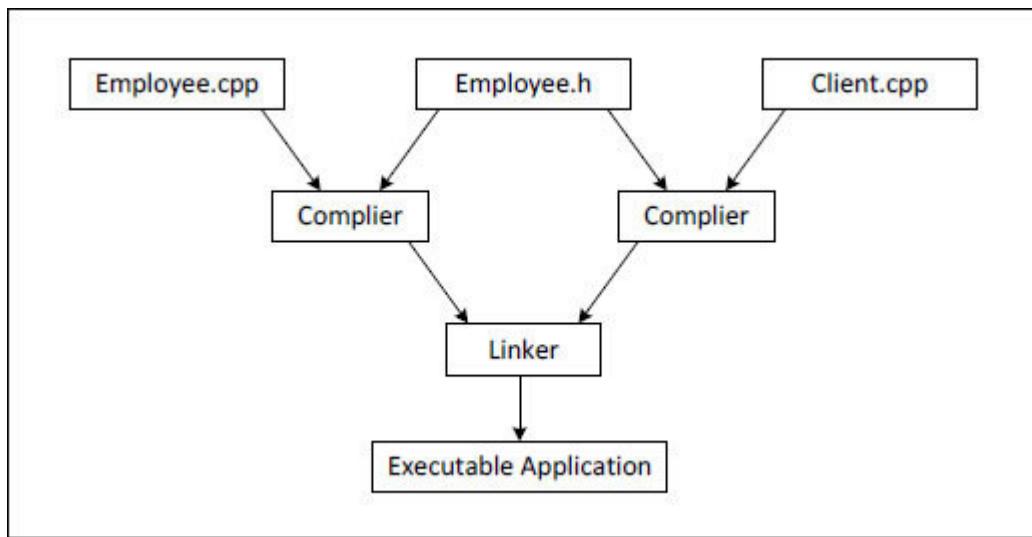


Figure *Compilation and Linking process.*

Exercise

[A] State True or False:

By default members of a structure are **public** and that of a class are private.

In a class data members are always whereas, member functions are always

It is necessary that a constructor in a class should always be

If a class contains a 3-argument constructor then it is necessary to define explicitly a zero-argument, a 1-argument and a 2-argument constructor.

Member functions of a class have to be called explicitly, whereas, the constructor gets called automatically.

A constructor gets called whenever an object gets instantiated.

Constructors can be overloaded.

A constructor never returns a value.

Size of an object is equal to sum of sizes of data members and member functions within the class.

If the `++` operator has been overloaded then the expressions `j++` and `++j` would call the same overloaded function.

When an object goes out of scope, its destructor gets called automatically.

If the binary `+` operator is overloaded inside a class then while calling it only one argument needs to be passed.

The `this` pointer always contains the address of the object using which the member function/data is being accessed.

The `this` pointer can be used even outside the class.

If member functions of a class are defined outside the class it is necessary to declare them inside the class.

[B] Point out the errors, if any, in the following programs.

```
(a) #include
using namespace std;
class Address
{
private:
string name, city;
public:
Address (string p, string q)
{
name = p;
city = q;
}
int main()
{
Address my ("Mac", "London");
return 0;
}
```

```
(b) class Date
{
private:
int day, month, year;
Date()
{
```

```
day = 7;  
month = 9;  
year = 2019;  
}  
};  
int main()  
{  
Date today;  
return 0;  
}
```

(c) class Value

```
{  
private:  
int i;  
float f;  
public:  
Value()  
{  
i = 0;  
f = 0.0;  
return 1;  
}  
};  
int main()  
{  
Value v1;
```

```
return o;
}
```

(d) #include

```
using namespace std;
class Triplets
{
private:
int t1, t2, t3;
public:
Triplets (int x, int y, int z)
{
t1 = x;
t2 = y;
t3 = z;
}
void display()
{
cout << endl << t1 << t2 << t3;
}
};

int main()
{
Triplets r (2, 3, 4), s;
r.display();
s.display();
return o;
```

```
}
```

```
(e) #include
using namespace std;
class Sample
{
private:
    int data1;
    float data2;
public:
    void Sample();
    void showData();
};

Sample::void Sample()
{
    data1 = 10;
    data2 = 20;
}

Sample::void showData()
{
    cout << endl << data1 << data2;
}

int main()
{
    Sample s;
```

```
s.showData();
return o;
}
```

```
(f) #include
using namespace std;
class List
{
private:
class Node
{
int data;
```

```
Node *link;
} *p;
public:
void create()
{
p = new Node;
p.data = 10;
p.link = NULL;
}
};
int main()
{
List l1;
l1.create();
return o;
```

```
}
```

[C] What will be the output of the following programs:

```
(a) #include
using namespace std;
class User
{
private:
int i;
float f;
char c;
public:
void displayData()
{
cout << i << '\n' << f << "\n" << c << endl;

}
};

int main()
{
cout << sizeof (User) << endl;
User u1;
cout << sizeof (u1) << endl;
u1.displayData();
return 0;
}
```

```
(b) #include
using namespace std;
class Date
{
private:
int dd, mm, yy;
public:
Date()
{
cout << "Reached here" << endl;
}
};

int main()
{
Date today;
Date *p = &today;
cout << p << endl;
return 0;
}
```

```
(c) #include
using namespace std;
class Student
{
private:
```

```
int m1, m2, m3;
float per;
public:
Student()
{
m1 = m2 = m3 = 0;
per = 0.0;
}
void calcPer (int x, int y, int z)
{
m1 = x; m2 = y; m3 = z;
per = (m1 + m2 + m3) / 3.0;
displayPer();
}
void displayPer()
{
cout << "Percentage =" << per << "%" << endl;
}
};

int main()
{
Student s1;
s1.displayPer();
s1.calcPer (35, 35, 35);

s1.displayPer();
return 0;
}
```

```
(d) #include
using namespace std;
class Control
{
public:
Control()
{
calculate();
cout << "Constructor" << endl;
}
void calculate()
{
display();
cout << "Calculator" << endl;
}
void display()
{
cout << "displayed" << endl;
}
};

int main()
{
Control c1;
return 0;
}
```

[D] Answer the following:

Modify the class **Rectangle** discussed in the text of this chapter such that a statement,

Rectangle r1 = 3;

assigns a value 3 to **len** as well as

If there are four objects used in a program how many **this** pointers would exist for these objects and why?

Define a **Matrix** class containing a 3×3 matrix. Define overloaded operators int this class to carry out the addition, multiplication and comparison of two matrices. Create Matrix objects in **main()** and call the overloaded functions to perform the matrix operations on them.

Define a **Time** class containing hours and minutes as private members and an overloaded **++** operator function. Create objects of this class in **main()** and call the operator function **++** to increment the time by 1 minute.

KanNotes

Structured programming encourages interaction of functions

Classes are user-defined data types

Classes indicate how the objects created from them would look like

Objects have specific data. Each object is a specific instance of a class

Data values in objects are often called instance data or state of the object

public members of a class are accessible from outside the class

private members of a class are NOT accessible from outside class

Within a class any member can access any other member

By default class members are private. By default structure members are public

Usually data in a class is kept private and the data is accessed / manipulated through public member functions of the class

Two ways to initialize an object:

Method 1: Using member function like getData() / setData()

Benefit 1 - Data is protected from manipulation

Benefit 2 - Better validation as it is done at one place

Benefit 3 - Validation done by class designer

Method 2: Using special member function - Constructor

Benefit 1 - Program is better organized

Benefit 2 - Guaranteed initialization through constructor (Ctor)

When an object is created, space is allocated in memory and Ctor is called

Name of Ctor must be same as name of class

Ctor is a function

Ctor doesn't return any value

Ctor gets called automatically when an object is created

Ctor is called only once during entire lifetime of an object

Ctor can be overloaded

If we don't define a Ctor, compiler inserts a 0-arg Ctor

A class may have Ctor as well as setData()

Ctor - To initialize object

setData() - To modify object

A destructor method is called when an object is about to be destroyed

Overloaded pre-increment and post-increment operators are distinguished by passing a `o` to the post-increment operator function

this is a constant pointer and it cannot be modified during execution of the method

this pointer dies once control returns from the method

In principle every object has instance data and member functions

In practice each object has instance data, whereas member functions are shared amongst objects

Sharing is justified as from one object to another member functions are going to remain same

It is a good software development practice to store the declaration of a class in a header file and its implementation in a `.cpp` file

Class Intricacies

Just knowing what classes are, and how to create objects from them is not enough. Unless you know their intricacies, you would not be able to exploit the benefits that they offer.

Static and Dynamic Memory Allocation

Allocation of Arrays and Structures

Allocation of Objects

Static Members

The Multi-purpose *const*

Overloaded = and Copy Constructor

Data Conversion

Conversion between Built-in Types

Conversion between Built-in and User-defined Types

Conversion between Different User-defined Types

Conversion Routine in Source Object

Conversion Routine in Destination Object

Exercise

KanNotes

In the last chapter we understood the basics of classes and objects. Now we know that a class is a user-defined data type and an object is an instance of it. This chapter explores several nuances of classes and objects. These include different ways of creating objects, its contents, its copying, its conversion, etc. These subtleties would improve your understanding of classes and objects. Let us begin with memory allocation.

Static and Dynamic Memory Allocation

As we know, while creating variables / objects memory is allocated for them. C++ allows us to allocate this memory in two ways. These are

Static memory allocation

Dynamic memory allocation

In the first method space is allocated for variables/objects in a region of memory known as Since decision about how much space to allocate and where to allocate it is taken at the compilation stage this method is known as static memory allocation. Actual creation of variables / objects happens only at execution stage. The storage class of these variables / objects decides how long they live.

In the second method decision about how much space to allocate and the actual allocation both happens at execution stage. The allocation happens in a region of memory called In the C++ lexicon, heap is called **free** The entities created in heap are nameless, so they can be accessed only using their

addresses. Hence when space is allocated for them, their address is returned and assigned to a pointer. This pointer is created in stack. Unlike static memory allocation, we can delete the entities created in heap any time during execution.

Dynamic memory allocation is called ‘dynamic’ because it lets us control the allocation and de-allocation of memory for any built-in or user-defined type. Dynamic allocation is done using an operator called **new** and de-allocation using an operator called

Given below are few examples of how to statically and dynamically allocate built-in types.

```
// static memory allocation
int i;
float a;
char ch;
// dynamic memory allocation
int *ptrInt = new int;
float *ptrFloat = new float;
char *ptrChar = new char;

// use allocated entities
*ptrInt = 35;
*ptrFloat = 3.14;
```

```
*ptrChar = 'A';

// eliminate allocated entities
delete ptrInt;
delete ptrFloat;
delete ptrChar;
```

The **new** operator allocates memory for a built-in data type and returns the address of the allocated memory. We have assigned this address to a suitable pointer. To use the data type we have to access it using the pointer. The **delete** operator does the reverse of It returns to the free store the memory owned by the allocated entity.

Dynamic memory allocation offers more flexibility—we can purge dynamically created entities when we want using the **delete** operator. However, there is an overhead too. For every entity created dynamically in heap, a pointer pointing to it has to be created in the stack. So to manage an **int** we need 4 bytes on heap and a 4-byte pointer on stack. Contrasted with this, a statically created **int** occupies only 4 bytes in stack.

ALLOCATION OF ARRAYS AND STRUCTURES

Dynamic memory allocation shouldn't be used for creating small-sized entities due to the overhead mentioned above. Instead, it should be used when entity size is big, as in the case of structures or arrays. This is because we can get rid of them when we are done with them. Following code snippet shows how to statically and dynamically allocate arrays and structures.

```
// structure declaration
struct Employee
{
    string name;
    int age;
    float salary;
};

// static memory allocation
int num[25];
struct Employee e;

// dynamic memory allocation
int *ptrInt = new int[25];
struct Employee *ptr = new struct Employee;

// use allocated entities
ptrInt [0] = 35;
ptrInt[20] = 40;
```

```
ptr->name = "Sanjay";
ptr->age = 23;
ptr->salary = 4500.00f;

// eliminate allocated entities
delete [] ptrInt;
delete ptr;
```

Note the usage of **new** to dynamically allocate an array. On allocating space for the array, base address of the array is returned, which is promptly assigned to Also note how the dynamically allocated array is eliminated from memory using the **delete** operator. The statement

```
delete [] ptrInt;
```

indicates that we are not deleting a **thing** but an array of being integers in this case) pointed to by the pointer Would a simple

```
delete ptrInt;
```

not work in this case? The compiler may not flag an error, but whether it would work successfully or not would vary from compiler to compiler. In some compilers the heap may get

corrupted, in some others only the first object in the array would get deleted. In short, you would be better off if you use **delete []** whenever we allocate memory using **new**

In statically allocated arrays we have to make a commitment to the size of the array at the time of writing the program. It means we cannot define an array like this

```
int n;  
cin >> n;  
int arr[n];
```

This is because the compiler has to decide the bytes to be allocated for this array at the time of compilation. At the time of compilation **n** doesn't have any value; we get the opportunity to supply value for **n** only during execution.

At all times, we may not know how many elements we propose to store in the array at the time of writing the program. As a result, often we end up either allocating too little or too much space for an array. This difficulty is eliminated in case of a dynamically allocated array as shown below.

```
int n;
```

```
cin >> n;  
int *p = new int[n];
```

Here based on the value of **n** supplied during execution, that big an array is created.

ALLOCATION OF OBJECTS

Dynamic allocation of objects has one additional aspect to it as compared to dynamic allocation of built-in types and arrays. When ‘objects’ are created dynamically in heap using the **new** operator three operations are performed by it. These are

It allocates storage of the proper size for an object

It runs the object's constructor function

It returns a pointer of the correct type

The fact that the constructor is executed guarantees proper initialization of object before we use it. Let us confirm this through a program.

```
#include  
using namespace std;
```

```
class Sample
{
private:
int i; float a;
public:
Sample()
{
    i = 10;
    a = 3.14f;
}
void showData()
{
    cout << i << endl << a << endl;
}
~Sample()
{
    cout << "Reached destructor" << endl;
}
};

int main()
{
    Sample *ptr;
    ptr = new Sample();
    ptr->showData();
    delete ptr;
}
```

Here is the output of the program

```
10  
3.14  
Reached destructor
```

The output clearly shows that the constructor gets called when we create the object dynamically using and the destructor gets called when we eliminate it using

Before we cover any fresh material here are a few subtleties of **new** and **delete** which would help you to understand them better.

An object created using **new** exists until it is explicitly destroyed using

The **delete** operator may be applied only to a pointer returned by **new** or to zero. Applying **delete** to **0** has no effect. In other words passing a NULL pointer to **delete** is safe and is guaranteed to do nothing. This simplifies the code that uses **delete** by allowing such code to **delete** a pointer that may be NULL without being required to check it using an

The expression **delete p** doesn't delete the pointer. It deletes the object being pointed to by **p**. So ideally the name of the keyword should have been **deletethethingpointedtoby** rather than **Any takers?**

Never delete a pointer twice. Suppose you have a pointer variable. The first time you **delete** the object ***p** is safely destructed, and the memory pointed to by **p** is safely returned to the heap. The second time you pass the same pointer to **delete** the remains of what used to be an object at ***p** are passed to the destructor (which could be disastrous), and the memory pointed to by **p** is handed back to the heap a second time. This may corrupt the heap.

Not deleting an object is not an error as far as the language is concerned. However, it is a bad practice. If the program is going to run for a long time then it is all the more important that we delete the object once its purpose is over.

If the class has a pointer as a data member and we initialize it in the constructor using **new** we must remember to **delete** it in the destructor. If we forget to do so a memory will leak. It means that the memory will remain allocated but there will be no way to access it.

Static Members

We know that each object contains its own set of data members, whereas, the member functions are shared amongst all objects. However, if we wish that a data member should be shared amongst all objects of that class it is possible. We simply need to declare it as Let me give you an example where a need for sharing a data member amongst all objects might be felt.

Suppose we have a **Circle** class and we wish to keep track of how many objects have been created from this class. For this we can declare a **static count** variable in the class. Every time a new object is created we can increment the value of **count** in the constructor. So its latest value will tell us how many objects of **Circle** class have been created so far. This is shown in the following program:

```
#include  
using namespace std;  
  
class Circle  
{
```

```
private:  
char color;  
float radius;  
static int count;  
public:  
Circle (char c, float r)  
{  
color = c;  
radius = r;  
count++;  
}  
  
static void showCount()  
{  
cout << "count =" << count << endl;  
}  
};  
  
int Circle::count = 0; // definition of count  
  
int main()  
{  
Circle c1 ('R', 1.2f);  
Circle::showCount();  
Circle c2 ('G', 2.2f);  
Circle::showCount();  
Circle c3 ('B', 3.2f);
```

```
Circle::showCount();  
return o;  
}
```

The class **Circle** has three data members, **radius** and **color**. Of these, the first two are instance data members, whereas, the third is a **static** data member. In **main()** we have defined three objects of class **Circle**. Each time an object is created we set up the **color** and **radius** of the **Circle** object and increment the value of **count**. Hence, **count** will get incremented thrice. The static member function **showData()** displays the current value of **count**. Here is the output of the program.

```
count = 1  
count = 2  
count = 3
```

As we expected, the value of **count** is incremented each time a new **Circle** object is created.

Observe carefully that **count** has been declared inside the class but defined outside it. Why such an approach is used for **static** data members? If **static** data members were defined inside the class declaration it would violate the idea that a class declaration is only a blueprint and does not reserve any memory.

A word of caution! If you include the declaration of a **static** variable but forget its definition, the compiler will pass it, whereas, the linker will report that you're trying to reference an unresolved external variable.

Also note the syntax of calling the **static** member function

```
Circle::showCount();
```

This shows that we are not calling it using an object. This is natural as **count** that is displayed in **showCount()** is not specific to any object.

Here are a few additional tips about static members...

An **instance** member function (like constructor) can access **instance** as well as **static** data members.

A **static** member function can access only **static** data members.

A **this** pointer never exists for a **static** function.

A **static** data member has nothing to do with **static** storage class.

This idea of sharing a variable across objects can be used in creating a software licensing system. Suppose a software company sells to its client a 5-seat license of its software. Every time a new user uses the software a count can be incremented. When user tries to use it, a violation of license terms can be reported.

The Multi-purpose const

In [Chapter 2](#) we saw usage of **const** keyword in two cases:

To prevent variables from being modified.

To prevent a function from modifying actual arguments using formal arguments.

We are allowed to use **const** in three more situations:

On objects

On member functions of a class

On member function arguments

The following program shows how to use **const** and effect that it has in these three cases.

```
#include  
using namespace std;
```

```
class Sample
{
private:
int data;
public:
Sample()
{
data = 0;
}
void changeData() const
{
data = 10;
}
void showData()
{
cout << endl << "data =" << data << endl;
}
void add (Sample const &s, Sample const &t)
{
data = s.data + t.data;
s.data = 45; // error
t.data = 50; // error
}
void getData()
{
cin >> data;
```

```
}

};

int main()
{
const Sample s1;
s1.getData();
// error Sample s2;
s2.changeData();
Sample s3;
s3.changeData();
Sample s4;
s4.add (s2, s3);
s4.showData();
return 0;
}
```

We have marked object **s1** as **const** so when we attempt to call **modifyData()** to modify it, compiler reports an error. We cannot change **s1** through any member function.

Though we haven't marked object **s2** as **const** we cannot modify it through **This** is because we have marked this member function as **A const** member function guarantees that it will never modify any of its class's member data. Note that unlike **s2** can be changed through other member functions like

It follows that you can use only **const** member functions with **const** objects, because they're the only ones that guarantee not to modify the object.

Note that to make **changeData()** constant the keyword **const** is placed after the declarator but before the function body. If the function is declared inside the class but defined outside it then it is necessary to use **const** in declaration as well as definition.

Lastly, in **add()** we have marked **s** and **t** as references to **const Sample** objects. As a result, when we attempt to change objects **s2** and **s3** through references **s** and compiler reports an error. This is indeed good, because during addition there is no reason why **s2** and **s3** should change.

Overloaded = and Copy Constructor

We know that C++ compiler provides a 0-argument constructor and a destructor if we don't provide it. Likewise, it also provides an overloaded assignment operator function and the copy constructor, if we don't provide them. Let us understand these two new functions. Consider the following statements:

```
Circle c1, c2;  
c1 = c2; // calls overloaded assignment operator function  
Circle c3 = c2; // calls copy constructor
```

Here **c1** and **c2** are objects of the type **Circle** which is a predefined class. The statement

```
c1 = c2;
```

will cause the compiler to copy the data from member-by-member, into This is what the **overloaded assignment operator** does by default.

In the next statement we have initialized one object with another object while defining it. This statement causes a

similar action. A new object **c3** is created and data from **c2** is copied member-by-member, into This is what the **copy constructor** does by default.

Note that in both statements cases we are using = but different functions get called. Why does copy constructor not get called in **c1** = That is because for **c1** a 0-argument constructor has already been called while defining it, and a constructor can be called for an object only once.

Likewise, why does overloaded assignment operator function not get called in **Circle c3** = That is because **c3** is yet to be created and assignment operator function can only be called for an existing object. Moreover, since the object **c3** is being defined, a constructor has to be called for it. This cannot be the normal constructor, since the object also has to be initialized with values from Hence a special constructor called copy constructor is called.

If you want that the assignment operator or the copy constructor should do something different than what's done by the compiler provided versions then you can always override them. This is shown in the following program.

```
#include  
using namespace std;
```

```
class Circle
{
private:
int radius;
float x, y;
public:
Circle()
{
}
Circle (int rr, float xx, float yy)
{
radius = rr;
x = xx;
y = yy;
}
Circle& operator = (Circle& c)
{
cout << "Assignment operator invoked" << endl;

radius = c.radius;
x = c.x;
y = c.y;
return *this;
}
Circle (Circle& c)
{
cout << "Copy constructor invoked" << endl;
```

```
radius = c.radius;
x = c.x;
y = c.y;
}
void showData()
{
cout << "Radius =" << radius << endl;
cout << "X-Coordinate =" << x << endl;
cout << "Y-Coordinate =" << y << endl << endl;
}
};

int main()
{
Circle c1 (10, 2.5f, 2.5f);
Circle c2, c3;
c3 = c2 = c1;
Circle c4 = c1;
c1.showData();
c2.showData();
c3.showData();
c4.showData();

return 0;
}
```

Here is the output of the program...

Assignment operator invoked

Assignment operator invoked

Copy constructor invoked

Radius = 10

X-Coordinate = 2.5

Y-Coordinate = 2.5

Radius = 10

X-Coordinate = 2.5

Y-Coordinate = 2.5

Radius = 10

X-Coordinate = 2.5

Y-Coordinate = 2.5

Radius = 10

X-Coordinate = 2.5

Y-Coordinate = 2.5

Most of the program is straightforward. What is important here is the function **operator =** which overloads the = operator, and the copy constructor.

The overloaded operator function gets called when the statement **c3 = c2 = c1** gets executed. Since there are two assignments here, the assignment operator is executed twice. That's why the message "Assignment operator invoked" gets

printed twice. The overloaded = operator does the copying of the member data from one object to another.

Note that we have passed the argument to overloaded operator function by reference. This is often desirable, though not absolutely necessary. Had the argument been passed by value it would have created another local object in the function. In our program it would not have mattered much, but in case of large objects this would lead to considerable wastage of memory.

Consider the statement,

```
c3 = c2 = c1;
```

As we know, during execution firstly **c2 = c1** will get executed. This internally becomes,

```
c2.operator = (c1);
```

The argument **c1** passed to the assignment operator function by reference. So reference **c** in this function refers to **c1**. Now data of **c1** is copied into **c2**'s data members through the statements,

```
radius = c.radius;  
x = c.x;  
y = c.y;
```

At this stage the **this** pointer contains address. Hence on returning ***this** we are simply returning. Moreover, it is being returned by reference. So another copy of it is not created. So next assignment becomes **c3 = *this**. This proceeds in the same way as discussed above.

Let us now turn our attention to the copy constructor. When the statement **Circle c4 = c1** gets executed, the overloaded copy constructor gets called. The copy constructor takes one argument, an object of the type passed by reference. Here's its prototype:

```
Circle (Circle &);
```

Is it necessary for us to use a reference in the argument to the copy constructor? Can we not pass a value instead? No. Because, if we pass the argument by value, its copy is constructed using the copy constructor. This means the copy constructor would call itself to make this copy. This process would go on and on until the compiler runs out of memory. Hence in the copy constructor the argument must always be passed by reference.

A copy constructor also gets invoked when objects are passed by value to functions and when objects are returned from functions. When an object is passed by value the copy that the function operates on is created using a copy constructor. If we pass the address or reference of the object the copy constructor would of course not be invoked, since in these cases the copies of the objects are not to be created.

When an object is returned from a function the copy constructor is invoked to create a copy of the value returned by the function.

Data Conversion

There are three different types of data conversion that may exist. These are

Conversion between built-in types

Conversion between built-in and user-defined types

Conversion between different user-defined types All these conversion are discussed below.

CONVERSION BETWEEN BUILT-IN TYPES

Consider the following code snippet containing operations on built-in types:

```
float a = 30; // int to double  
double b = 1.55f; // float to double  
int a = 3.14; // double to int  
float b = 6.28; // double to float
```

On compilation the first two assignments are passed since those are widening conversions. However, warnings are reported for the next two since they are narrowing conversions—conversions where loss of precision is likely to happen. To eliminate the warnings we can use typecasting as shown below.

```
int a = int (3.14); // double to int  
float b = float (6.28); // double to float
```

Widening conversions happen implicitly. For narrowing conversions, we need to do typecasting.

CONVERSION BETWEEN BUILT-IN AND USER-DEFINED TYPES

When we want to convert between user-defined data types and basic types, we can't rely on built-in conversion routines, since the compiler doesn't know anything about user-defined types besides what we tell it. Instead, we must write these conversion routines ourselves.

The following program shows how to convert between a basic type and a user-defined type and vice versa. In this program the user-defined type is a **string** class and the basic type is

The program shows conversion from **string** to **int** and from **int** to Here's the program...

```
// conversions: String to int, int to String
#include

#include
using namespace std;

class String
{
private:
char str[20];
public:
String()
{
str[0] = '\0';
}
String (char *s)
{
strcpy (str, s);
}
String (int a)
{
itoa (a, str, 10);
}
```

```
operator int()
{
int i = 0, l, num = 0, k = 1;
l = strlen (str) - 1;
while (l >= 0)
{
num = num + (str[l] - 48) * k;
l--;
k *= 10;
}
return (num);
}
void displayData()
{
cout << str << endl;
}
};
```

```
int main()
{
String s1 = 123;
cout << "s1 =";
s1.displayData();
```

```
s1 = 150;
cout << "s1 =";
s1.displayData();
```

```
String s2 ("123");
int i = int (s2);
cout << "i =" << i << endl;

String s3 ("456");

i = s3;
cout << "i =" << i << endl;

return o;
}
```

In this program to convert an **int** to a user-defined type **String** we have used a constructor with one argument. It is called when an object of type **String** is created with a single argument as in

```
String s1 = 123; // this is same as String s1 (123);
```

The constructor converts the **int** to a string and assigns it to **str** using the **itoa()** function.

In the statement

```
s1 = 150;
```

we are converting an **int** to a but we are not creating a new object. The one argument constructor is called even in this case. When the compiler comes across a statement that needs a conversion, it looks for any tool that can carry out this work for it. In our program it finds a constructor that converts an **int** to a so it uses it in the assignment statement by first creating an unnamed temporary object with its **str** holding the value corresponding to the integer **150** and then assigns this object to Thus if the compiler doesn't find an overloaded = operator it looks for a constructor to do the same job.

To convert a **String** to an **int** the overloaded cast operator is used. This is often called a conversion function. This operator takes the value of the **string** object of which it is a member, converts this value to an **int** value and then returns this **int** value. This operator gets called in two cases:

```
i = int (s2);
```

and

```
i = s3;
```

In the second assignment the compiler first searches for an overloaded assignment operator. Since this search fails the compiler uses the conversion function to do the job of conversion.

One might think that it would not be a sound programming practice to routinely convert from one type to another. However, the flexibility provided by allowing conversions often outweighs the dangers of making mistakes by allowing mixing of data types.

CONVERSION BETWEEN DIFFERENT USER-DEFINED TYPES

As in the case of conversion between user-defined and built-in data types, for converting data between objects of different user-defined classes the same two methods are used—a one-argument constructor, or a conversion function. The choice depends on where we want to put the conversion routine—in the class declaration of the source object or of the destination object. We propose to examine both the cases here.

CONVERSION ROUTINE IN SOURCE OBJECT

When the conversion routine is in the source class, it is commonly implemented as a conversion function as shown in

the following program. The two classes used in the program are **Date** and **dmy**. Both classes are built to handle dates, the difference being the **Date** class handles it as a string, whereas the **dmy** class handles it as three integers representing day, month and year. Here is the listing of the program...

```
#include
#include
using namespace std;

class Date
{
private:
char dt[9];
public:
Date()
{
dt[0] = '\0';
}
Date (char *s)
{
strcpy (dt, s);
}
void displayData()
{
cout << dt << endl;
```

```
}

};

class dmy
{
private:
int day, mth, yr;
public:
dmy()
{
day = mth = yr = 0;

}
dmy (int d, int m, int y)
{
day = d;
mth = m;
yr = y;
}
operator Date()
{
char temp[3], str[9];
itoa (day, str, 10);
strcat (str, "/");
itoa (mth, temp, 10);
strcat (str, temp);
strcat (str, "/");
itoa (yr, temp, 10);
```

```

        strcat (str, temp);
        return (Date (str));
    }
    void displayData()
    {
        cout << day << "/" << mth << "/" << yr << endl;
    }
};

int main()
{
    Date d1;
    dmy d2 (17, 11, 94);
    d1 = d2;

    cout << "d1 =";
    d1.displayData();
    cout << "d2 =";
    d2.displayData();
    return 0;
}

```

In **main()** we have defined an object **d1** of the type which is not initialized. We have also defined an object **d2** of the type which has been initialized. Next an assignment is carried out through the statement **d1 =**

Since **d1** and **d2** are objects of different classes, the assignment involves a conversion, and as we specified, in this program the conversion function **Date()** is a member of the **dmy** class, i.e. the source class. This function transforms the object of which it is a member to a **Date** object, and returns this object, which **main()** then assigns to

CONVERSION ROUTINE IN DESTINATION OBJECT

Let's now see how the same conversion is carried out when the conversion routine is present in the destination class. In such cases usually a one-argument constructor is used. However, things are complicated by the fact that the constructor in the destination class must be able to access the data in the source class to perform the conversion. That is, since the data members **mth** and **yr** in the **dmy** class are **private** we must provide functions **getDay()**, **getMth()** and **getYr()** to return them. Here's a program that implements this.

```
#include  
#include  
  
using namespace std;  
  
class dmy
```

```
{  
private:  
int day, mth, yr;  
public:  
dmy()  
{  
day = mth = yr = 0;  
}  
dmy (int d, int m, int y)  
{  
day = d;  
mth = m;  
yr = y;  
}  
int getDay()  
{  
return (day);  
}  
int getMth()  
{  
return (mth);  
}  
int getYr()  
{  
return (yr);  
}  
void displayData()
```

```
{  
cout << day << "/" << mth << "/" << yr << endl;  
}  
};
```

```
class Date  
{  
private:  
char dt[9];  
public:  
Date()  
{  
dt[0] = '\0';  
}  
Date (char *s)  
{  
strcpy (dt, s);  
}  
void displayData()  
{  
cout << dt << endl;  
}  
Date (dmy t)  
{  
int d = t.getDay();  
int m = t.getMonth();  
int y = t.getYear();  
}
```

```
char temp[3];
itoa (d, dt, 10);
strcat (dt, "/");
itoa (m, temp, 10);
strcat (dt, temp);
strcat (dt, "/");
itoa (y, temp, 10);
strcat (dt, temp);
}
```

```
};

int main()
{
Date d1;
dmy d2 (17, 11, 19);
d1 = d2;
cout << "d1 =";
d1.displayData();
cout << "d2 =";
d2.displayData();
return 0;
}
```

When we execute the statement **d1 = d2** the one-argument constructor in the **Date** class (whose argument is a **dmy** object) gets called. This constructor function gets the access

to the data of `d2` by calling the `getMth()` and `getYr()` functions. Finally it converts this data into a string. The output of this program is similar to the earlier one. The difference is behind the scenes. Here a constructor in the destination object, rather than a conversion function in the source object, handles the conversion.

That brings us to the important question—when should we use a one- argument constructor in the destination class, and when should we use a conversion function in the source class? Often this choice is simple. If you have a library of classes, you may not have access to its source code. If you use an object of such a class as the source in a conversion, then you'll have access only to the destination class, and you'll need to use a one-argument constructor. Or, if the library class object is the destination, then you must use a conversion function in the source. What if we use a conversion function as well as a one-argument constructor? The compiler will report an error since this becomes an ambiguous situation.

Exercise

[A] State whether the following statements are True or False:

Static memory allocation takes place during compilation, whereas, dynamic memory allocation takes place during execution.

If memory is allocated using `new []` it must be de-allocated using `delete []`.

`new` not only allocates memory but also calls the object's constructor.

Heap and free store are two different things.

a NULL pointer is safe and is guaranteed to do nothing.

In C++ to reallocate memory we should use the `renew` operator.

The **new** operator always returns a **void** pointer which needs to be typecasted explicitly.

Like other operators the **new** operator can also be overloaded.

We should never **delete** a pointer twice.

A **static** data member is useful when all objects of the same class must share a common item of information.

If a class has a **static** data member and three objects are created from this class, then each object would have its own **static** data member.

A class can have **static** data members as well as **static** member functions.

A **static** data member's definition appears in the class declaration, but the variable is actually declared outside the class.

If **display()** is a **static** member function of a class called **sample** then it can be called in the following way:

```
Sample s1;
```

```
s1.display();
```

If **display()** is a **static** member function of a class called **sample** then it can be called in the following way:

```
Sample::display();
```

The **const** can be used on member functions of a class as well as on member function arguments.

A **const** member function prevents modification of any of its class's member data.

If a member function of a class is to be **const** then it is necessary to use **const** in declaration as well as definition of the member function.

You can use only **const** member functions with **const** objects.

If we don't provide an assignment operator in a class declaration then the compiler automatically adds one to our class.

If we don't provide a copy constructor in a class declaration then the compiler automatically adds one.

The following two set of statements are same:

```
Sample s1;  
s1 = s2;
```

and

```
Sample s1 = s2;
```

It is not possible to return a local object by reference.

When an object is passed to a function or returned from a function the copy constructor gets called.

To carry out conversion from an object to a basic type or vice versa it is necessary to provide the conversion functions.

To carry out conversion from object of one user-defined type to another it is necessary to provide the conversion functions.

[B] Answer the following:

Does the expression **delete p** delete the pointer or the object being pointed to by

Write a program that will allocate memory for a 1-D, 2-D and a 3-D array of integers. Store some values in these arrays and then print them out. We must be able to access elements of these arrays using forms **b[i][j]** and

How many bytes would be allocated by the following code?

```
int main()
{
const int MAXROW = 3;
const int MAXCOL = 4;
int (*p)[MAXCOL];
p = new int[MAXROW][MAXCOL];
return 0;
}
```

What will be the output of the following program?

```
#include
using namespace std;
int main()
{
```

```
const int MAXROW = 3;  
const int MAXCOL = 4;  
int (*p)[MAXCOL];  
p = new int[MAXROW][MAXCOL];  
cout << endl << sizeof (p) << endl << sizeof (*p);  
return 0;  
}
```

In which of the following cases can we use

- On normal variables
- On global function arguments
- On member functions of a class
- On member function arguments
- On objects

If we wish to provide an assignment operator and a copy constructor within a class called **Rectangle** what would be their prototypes?

What is the difference in the following two statements?

```
delete a;  
delete [] a;
```

What does the delete operator do in addition to de-allocating the memory used by the object?

Write a program that consists of two classes **time12** and **time24**. The first one maintains time on a 12-hour basis, whereas the other one maintains it on a 24-hour basis. Provide conversion functions to carry out the conversion from object of one type to another.

Write a program that implements a **date** class containing data members **month** and **year**. Implement overloaded assignment operator function and copy constructor in this class.

KanNotes

In static memory allocation decisions about how many bytes to allocate and where to allocate them is taken at compilation stage

In dynamic memory allocation decisions about how many bytes to allocate and where to allocate them is taken at execution stage

Stack and heap are regions in memory

Static memory allocation allocates space in stack

Dynamic memory allocation allocates space in heap

Heap is also called free store

It is a good practice to create normal variables in stack

It is a good practice to arrays, structures and objects in heap

If new is used in the constructor, use delete in the destructor

delete ptr deletes the entity pointed to by ptr

Memory allocated using new [] should be deleted using delete []

Memory leak occurs when memory stands allocated but we have no way to use it or de-allocate it

Static data members are shared amongst multiple objects

Static member functions can access only static data members

Static data member has to be declared inside the class but defined outside it

Static member function can be accessed using the syntax classname::functionname()

A variable, object, function arguments marked as const cannot be modified

Function marked as const cannot modify the object using which it is called

Compiler provides overloaded = operator function copy constructor if we do not provide them

For built-in types widening conversions happen implicitly, whereas narrowing conversion should be done using typecasting

Conversion from built-in type to user-defined type can be done using a constructor

Conversion from user-defined type to a built-in type can be done using overloaded typecast operator function

Conversion from one user-defined type to another can be done using a conversion function when the conversion function is in source object

Conversion from one user-defined type to another can be done using a constructor when the conversion function is in destination object

Inheritance

Technology thrives on reuse, rather than on reinvention. Reuse saves times, effort and money. Same is true about programming. C++ has several reuse mechanisms. Of these, Inheritance is perhaps the most novel. This chapter explains this mechanism through example programs.

Inheritance

Another Inheritance Example

Uses of Inheritance

Constructors in Inheritance

Types of Inheritance

Simple Inheritance

Multi-level Inheritance

Multiple Inheritance

A Word of Caution

Incremental Development

Exercise

KanNotes

Now that we are more than familiar with classes and objects—the building blocks of object oriented programming—let us now deal with another important C++ concept called reuse mechanisms. C++ offers three reuse mechanisms. They are

Inheritance

Containership

Templates

In all three mechanisms we can reuse existing classes and create new enhanced classes based on them. To use the template reuse mechanism it is mandatory that the existing class's source code must be available. As against this, containership and inheritance can reuse existing classes even if their object code is not available. In this chapter we are going to discuss the inheritance mechanism alone.

Inheritance

Inheritance is the process of creating new classes, called **derived** from existing classes. The existing classes are often called **base**. The derived class inherits all the capabilities of the base class but can add new features and refinements of its own. By adding these refinements the base class remains unchanged.

Once a base class is written and debugged, it need not be touched again but at the same time it can be adapted to work in different situations. Reusing existing code saves time and money and increases a program's reliability. Inheritance can also help in the original conceptualization of a programming problem, and in the overall design of the program.

The code reusability is of great help in the case of distributing class libraries. A programmer can use a class created by others, and, without modifying it, derive other classes from it that are suited to particular programming situations.

Let us now understand the concept of inheritance using a program. Suppose that we have designed a class called **Index** that serves as a general-purpose index (counter). Assume that we have worked long and hard to make the **Index** class operate just the way we want, and we're pleased with the results, except for one thing. The **Index** class can only increment the counter and not decrement it.

To achieve this we can insert a decrement function directly into the source code of the **Index** class. However, there are several reasons why we might not want to do this. Firstly, the **Index** class is working well and has been thoroughly tested and debugged. This is an exaggeration in this case, but it would be true in a larger and more complex class. Now if we start modifying the source code of the **Index** class, the testing process will need to be carried out again. And then there always exists a possibility that at the end of the entire process the original class itself may not work satisfactorily.

Sometimes there might be another reason for not modifying the **Index** class—we might not have access to its source code, especially if it had been distributed as part of a class library.

To avoid these problems we can use inheritance to create a new class based on without modifying **Index** itself. Here's how this can be achieved.

```
#include
using namespace std;

class Index // base class
{
protected:
int count;
public:
Index()
{
count = 0;
}
void display()
{
cout << "count =" << count << endl;
}
void operator ++ ()
{
count++;
}

};

class NewIndex: public Index // derived class
{
public:
```

```
void operator -- ()  
{  
    count--;  
}  
};  
  
int main()  
{  
    NewIndex i;  
    ++i;  
    ++i;  
    i.display();  
    --i;  
    i.display();  
    return 0;  
}
```

Here we have first declared a base class called **Index** and then derived a class called **NewIndex** from it. **NewIndex** inherits all the features of the base class. It doesn't need the **operator ++ ()** and **display()** functions, since they are already present in the base class.

The first line of the **NewIndex** class,

```
class NewIndex: public Index
```

specifies that the class **NewIndex** has been derived from the base class We would see later on what happens if we use **private** in place of [Figure 6-1](#) shows the relationship between the base class and the derived class.

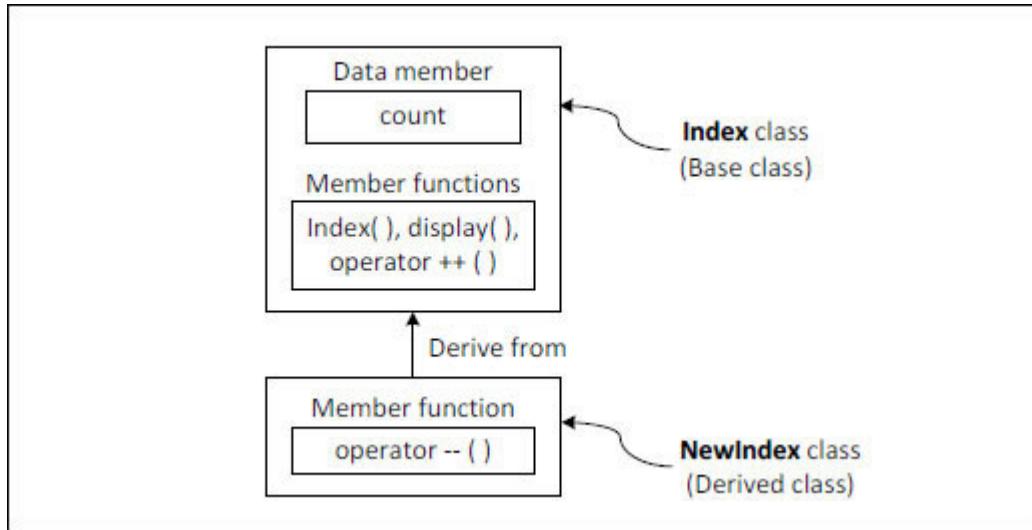


Figure 6-1. Inheritance.

Note that the arrow in the figure means **derived** The direction of the arrow says that the derived class refers to the functions and data in the base class, while the base class has no access to the derived class data or functions.

Since we have not declared any constructor in **NewIndex** class, compiler will insert a 0-argument constructor in it. When we create a **NewIndex** object the constructor in the base class as well as the one in the derived class will get called, in that order.

When we call **operator ++()** or **display()** function using they are first searched in **NewIndex** class. Since they are not found in the search is continued in its base class They are found in **Index** and are executed from there.

When we call **operator --()** using it is first searched in **NewIndex** class. It is found in hence it is executed from there to decrement

The data members in the classes that we've created so far used the **private** access specifier. However, in this program we have marked count as a **protected** data member. Had it been marked it would not have been available to **operator --()** since this function lies outside the Index class. We didn't want to make **count** since that would allow it to be accessed through any function anywhere in the program, and thereby eliminate the advantages of data hiding.

A **protected** member, on the other hand, can be accessed by member functions in its own class or in any class derived from its own class. It can't be accessed from functions outside these classes, such as This is just what we want. Hence the **protected** access specifier.

[Figure 6-2](#) clearly indicates who can access what in a base class-derived class relationship. Instead of the **base - derived** terminology some authors use the **parent - child** terminology.

You would agree that through the derived class **NewIndex** (and thereby through inheritance) we have increased the functionality of the **Index** class without modifying it.

Note that inheritance doesn't work in reverse. That is, the base class and its objects have no knowledge about any classes derived from the base class. In our program had we built an object **j** from the class then the **operator --()** function would have remained inaccessible to this object.

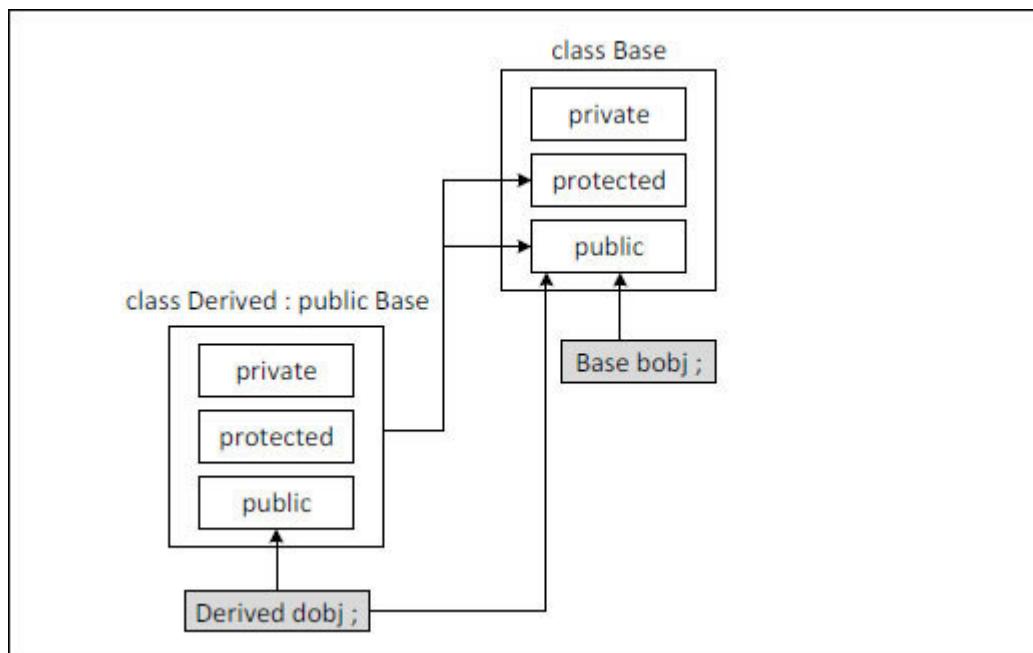


Figure 6-2. Access in inheritance.

Note that a program can declare objects of both the base and derived classes. The two objects are independent of one another.

Another Inheritance Example

Let us now look at another example that puts inheritance to work. Go through it and understand the basic concept of inheritance. In the next few pages we'll examine a few more features of inheritance in more detail.

```
#include
using namespace std;

const int MAX = 10; class Stack // base class
{
protected:
int arr[MAX];
int top;
public:
Stack()
{
top = -1;
}
void push (int num)
{
top++;
arr[top] = num;
```

```
}

int pop()
{
    int num;
    num = arr[top];
    top--;
    return (num);
}

};

class NewStack: public Stack // derived class
{
public:
bool isFull()
{
    if (top == MAX - 1)
        return true;
    else
        return false;
}
bool isEmpty()
{
    if (top == -1)
        return true;
    else
        return false;
}
};
```

```
int main()
{
    NewStack stk;

    if (!stk.isEmpty())
        stk.push (10);
    else
        cout << "Stack is full" << endl;

    if (!stk.isEmpty())
        stk.push (20);
    else

        cout << "Stack is full" << endl;
    if (!stk.isEmpty())
        stk.push (30);
    else
        cout << "Stack is full" << endl;

    int n;
    if (!stk.isEmpty())
    {
        n = stk.pop();
        cout << "Item popped =" << n << endl;
    }
    else
```

```
cout << "Stack is empty" << endl;

if (!stk.isEmpty())
{
n = stk.pop();
cout << "Item popped =" << n << endl;
}
else
cout << "Stack is empty" << endl;

if (!stk.isEmpty())
{
n = stk.pop();
cout << "Item popped =" << n << endl;
}
else
cout << "Stack is empty" << endl;
if (!stk.isEmpty())
{
n = stk.pop();
cout << "Item popped =" << n << endl;
}
else
cout << "Stack is empty" << endl;
return o;
}
```

As you might be aware, the **stack** data structure is a ‘Last In First Out’ list. It is implemented in the base class. It has the capability to push items on the stack and pop items off it. However, the function **Stack::push()** doesn't consider the possibility of the stack becoming full. Similarly, the **Stack::pop()** function doesn't consider the possibility that at some point the stack may fall empty and it cannot remove an element from an empty stack.

To account for these possibilities we have derived a class **NewStack** from **Stack**. Within it we have provided two functions— **isFull()** and **isEmpty()**. These functions report whether stack is full or not and whether it is empty or not, respectively.

Note that in **main()** we have created a derived class object. Through it, not only **isFull()** and **push()** and **pop()** of **Stack** class are also accessible.

Thus using inheritance we have provided additional functionality to the stack without modifying the base class.

Uses of Inheritance

Now that we have a basic idea about inheritance let us see in which scenarios is inheritance used in C++. The four common usages of inheritance are as follows:

Use existing functionality

Override existing functionality

Provide new functionality

Combination of existing and new functionality

Let us now look at a few examples of these usage patterns. We would begin with a scenario which demonstrates usage of all the four features mentioned above. Here is the program...

```
// Program demonstrating various Inheritance usage scenarios
#include
using namespace std;
```

```
class Ex
{
public:
void fun()
{
cout << "Inside Ex - fun()" << endl;
}
void save()
{
cout << "Inside Ex - save()" << endl;
}
void enc()
{
cout << "Inside Ex - enc()" << endl;
}

void open()
{
cout << "Inside Ex - open()" << endl;
}
};

class NewEx: public Ex
{
public:
void save()
{
cout << "Inside NewEx - save()" << endl;
}
```

```
void enc()
{
cout << "Inside NewEx - enc()" << endl;
}

void autoUpdate()
{
cout << "Inside NewEx - autoUpdate()" << endl;
}

void open()
{
cout << "Inside NewEx - open()" << endl;
Ex::open();
}

};

int main()
{
NewEx e;

e.fun();
e.save();
e.enc();
e.autoUpdate();
e.open();
}
```

Given below is the output of the program...

```
Inside Ex - fun()
Inside NewEx - save()
Inside NewEx - enc()
Inside NewEx - autoUpdate()
Inside NewEx - open()
Inside Ex - open()
```

Here we have defined two classes —**Ex** as base class and **NewEx** as derived class. The **Ex** class contains member functions **enc()** and whereas **NewEx** class contains member functions **open()** and In **main()** we have created an object of **NewEx** class and then called different member functions. Now look at the output of the program to appreciate the different inheritance usage scenarios.

When we called the function **fun()** it was first searched in the **NewEx** class. Since it was not found in it was then searched in the base class. In the base class it was found. As a result, the base class function got called and the message “Inside Ex - Fun()” got printed. This demonstrates that we have used one of the features of the base class, namely as it is, through a derived class object.

When we called again the function was searched in derived class Since the function was found in this version of **save()**

(and not the one in **Ex** class) got called. This demonstrates that through inheritance we are able to override existing functionality. Same is the case with the function

When we called the function **autoUpdate()** the function was searched and found in Since there was no **autoUpdate()** function in base class this is the case where a totally new functionality is provided through inheritance.

Lastly, when we called **open()** the function was searched and found in This version of **open()** got called and this function in turn called the base class version of **open()** through the statement The resulting output was:

Inside NewEx - open()

Inside Ex - open()

This indicates the usage of inheritance to combine new functionality with old.

Constructors in Inheritance

As we saw in the last program, the base class member functions can be called from derived class member function using the syntax:

```
baseclassname::functionName();
```

Unless explicitly called, the base class function doesn't get called from the body of the derived class function.

Constructors in inheritance chain are given a different treatment. Regarding constructors in inheritance chain we have to keep two things in mind:

When a derived class object is created, the constructor of the base class followed by constructor of derived class gets called.

While constructing a derived class object, if we do not call the base class constructor then, by default, the zero-argument constructor of base class gets called.

Let us illustrate these facts with the help of a program. Here is the program...

```
// Demonstrates calls to constructors in Inheritance chain
#include
using namespace std;

class Base
{
public:
    Base()
    {
        cout << "Base's 0-arg Ctor" << endl;
    }

    Base (int xx)
    {
        cout << "Base's 1-arg Ctor" << endl;
    }
};

class Der: public Base
{
public:
    Der()
    {
        cout << "Der's 0-arg Ctor" << endl;
    }

    Der (int x): Base (x)
    {
    }
```

```
cout << "Der's 1-arg Ctor" << endl;
}
};

int main()
{
Der y;
Der z (10);
}
```

Given below is the output of the program...

```
Base's 0-arg Ctor
Der's 0-arg Ctor
Base's 1-arg Ctor
Der's 1-arg Ctor
```

From the output we can see that when we construct the objects **y** and firstly the constructor of base class gets called, followed by constructor of the derived class. While constructing the object **y** the zero-argument constructor of base class got called automatically. However, while constructing **z** the one-argument constructor had to be called explicitly through the syntax:

Der (int x): Base (x)

Had we not used **Base** then the zero-argument constructor of base class would have been called.

This means that the parameters list for the derived class's constructor function may be different from that of the base class's constructor function. Therefore, the constructor function for the derived class must tell the compiler what values to use as arguments to the constructor function for the base class.

You must be wondering why it is necessary that the construction should proceed from base towards derived. This can be best understood with the help of the following program:

```
// Order of construction of object in Inheritance chain
class Base
{
protected:
int i;
public:
Base()
{
i = 4;
}
};

class Der: public Base
```

```
{  
  
private:  
int j;  
public:  
Der()  
{  
j = i * 4;  
}  
};  
int main()  
{  
Der d;  
}
```

Here **i** is marked as a **protected** member, hence it is available to functions in the derived class. For a moment, assume that the order of construction is not from base towards derived. In that case, firstly the constructor of **Der** would get called. And since **i** is available in constructor and the base class constructor has not been called so far, it has not been set with a value so far. As a result, **j** would not be set properly. Thus, the value in the object **d** would not be set correctly. Unlike, this if the order of construction is from base towards derived then object **d** would be set initialized properly.

In an inheritance relationship all functions do not get inherited. In particular, constructor, destructor, overloaded = operator function and copy constructor do not get inherited. In general these functions deal with initialization, assignment or destruction of an object, and they know what to do with the aspects of the object only at their particular level in the class hierarchy. Hence they are not inherited.

Types of Inheritance

C++ permits three different types of inheritance. These are

Simple inheritance

Multi-level inheritance

Multiple inheritance

Let us discuss each of these types.

SIMPLE INHERITANCE

All the programs in this chapter followed simple inheritance, where one class got derived from another. Even in this types of inheritance there are three sub-types:

Public inheritance

Protected inheritance

Private inheritance

All our programs used public inheritance through statements like

```
class Der: public Base  
class NewStack: public Stack
```

We can do protected or private inheritance through the statements

```
class Der: protected Base  
class NewStack: private Stack
```

If we inherit a class using the statement,

```
class Der: Base
```

then it is assumed that we are intending to do **private** inheritance.

Note that when we do **public** inheritance **public** and **protected** members remain **public** and **protected** in the derived class.

Unlike that, in **protected** inheritance **public** and **protected** members of base class become **protected** in the derived class. In **private** inheritance, **public** and **protected** members of base class become **private** in the derived class. This is shown in [Figure](#)

Can we inherit two classes from one base class? Of course, we can. For example, we may have a base class called **Shape** and from this we can inherit a **Circle** class and **Rectangle** calls by way of simple inheritance.

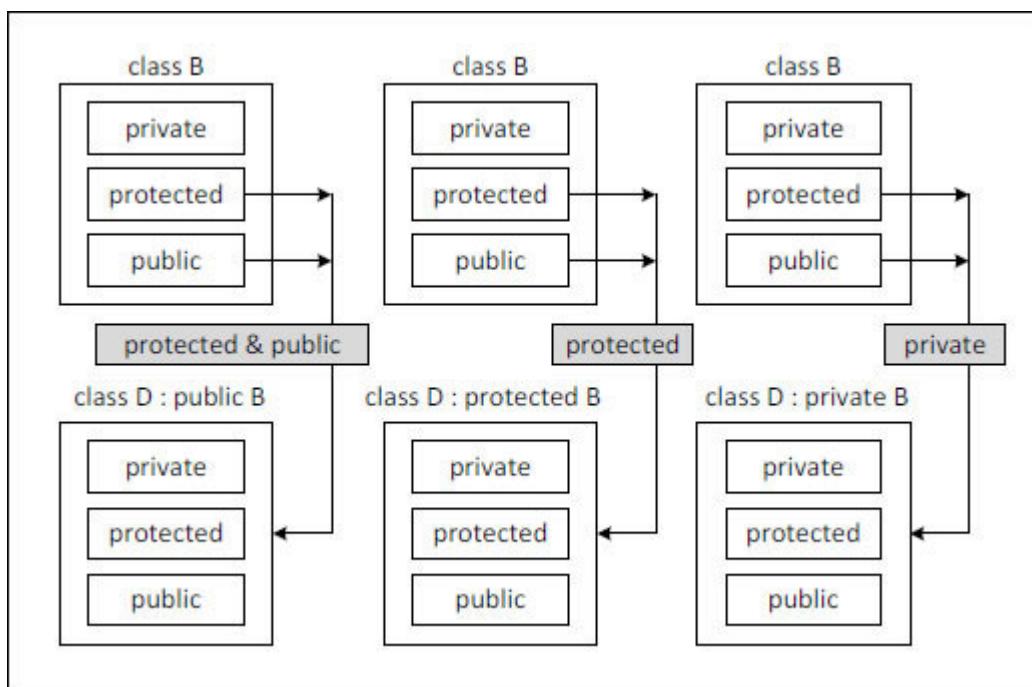


Figure 6-3. Types of simple inheritance.

Occasionally, there may be situations where you want to hide part of the functionality of the base class. For example,

suppose we want that a **Base** class function **display()** should be accessible from outside the class and rest of the functions should not be. To achieve this, we can do **private** inheritance and add the following statement in the derived class:

```
base:: display;
```

We do not use protected inheritance often; it's there in the language for the sake of completeness.

MULTI-LEVEL INHERITANCE

We can even derive a class from a class which itself has been derived from another class. Thus, multiple levels of inheritance can exist. For example, we may have a base class called **LinkedList** from which we derive a class Then we may derive a class **LinkedList2** from There is no limit on number of levels in multi-level inheritance.

MULTIPLE INHERITANCE

In multiple inheritance, a class is derived from more than one base class. Let us understand multiple inheritance using a simple example.

Suppose a company that markets both hardware and software. To implements its sales system, we can create four classes—**HardwareItem** and [Figure 6-4](#) displays the contents of these classes as well as the relationship between these classes.

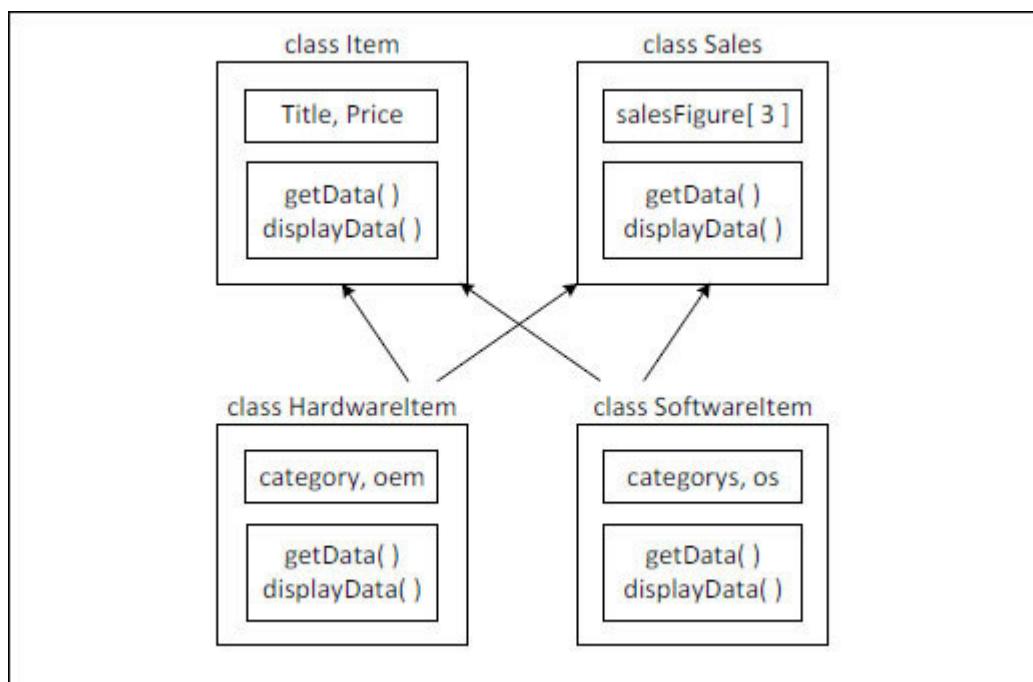


Figure 6-4. *Multiple inheritance.*

The declarations of these classes will be as under.

```
class Item
{
private:
    string title;
```

```
float price;
public:
void getData();
void displayData();
};

class Sales
{
private:
float salesFigure[3];
public:
void getData();

void displayData();
};

class HardwareItem: private Item, private Sales
{
private:
string category;
string oem;
public:
void getData();
void displayData();
};

class SoftwareItem: private Item, private Sales
{
private:
string category;
string os;
```

```
public:  
void getData();  
void displayData();  
};
```

Though most of these declarations are simple to understand, there are two points worth noting:

The declaration

```
class HardwareItem: public Item, public Sales
```

indicates that the class **HardwareItem** has been derived from two classes **Item** and **Sales**. Hence, it inherits the properties of both these classes.

The order of classes in this inheritance declaration is important. If we create an object of **HardwareItem** then the constructors will be called in the sequence. After that the body of **HardwareItem()** constructor will be executed. Same argument will hold good for multi-argument constructors.

If we call **displayData()** function from the **displayData()** function of the **HardwareItem** (derived class), the compiler would not know the **displayData()** function of which base class

or should be called. This ambiguity can be easily eliminated by preceding the **displayData()** function with the class name and the scope resolution operator as shown below:

```
Item :: displayData();  
Sales :: displayData();
```

A Word of Caution

Imagine a situation where a class has been derived from two base classes. Suppose the two base classes have functions with the same name, say while the derived class has no function with this name. What will happen if the derived class object tries to call the base class function? An error would occur since the compiler can't figure out which base class's **fun()** you wish to call. This problem can be resolved using the scope resolution operator as shown below:

```
// Base1 and Base2 are base classes
// Both base classes contain the function fun()
// The class Der has been derived from Base1 and Base2
Der d;
d.fun(); // Error
d.Base1::fun();
d.Base2::fun();
```

Incremental Development

One of the advantages of inheritance is that it supports incremental development. It allows you to introduce new code without causing bugs in existing code. By inheriting from an existing, functional class and adding data members and member functions (and redefining existing member functions) you leave the existing code—that someone else may still be using—untouched and bug-free. If a bug occurs, you know it's in your new code, which is much shorter and easier to read than if you had modified the body of existing code.

It's rather amazing how cleanly the classes are separated. You don't even need the source code for the member functions to reuse the code. Just the header file describing the class and the object file or library file with the compiled member functions would do.

It's important to realize that program development is an incremental process. Nobody ever conceived the program in its entirety at the start of the project. The program should try to create and manipulate objects of various types to express a model in the terms given to you by the problem's space. Rather than constructing the program all at once it should

grow out as an organic, evolutionary creature. Of course, at some point after things stabilize you need to take a fresh look at your class hierarchy with an aim to collapse it into a sensible structure. Inheritance fits this bill to perfection.

Exercise

[A] State True or False:

We can derive a class from a base class even if the base class's source code is not available.

Multiple inheritance is different from multiple levels of inheritance.

The way a derived class member function can access base class **protected** and **public** members, the base class member functions can access **protected** and public **member** functions of derived class.

It is possible to derive a derived class through **public** derivation, **private** derivation or **protected** derivation.

A derived class member function has an access to **protected** and **public** members of base class, irrespective of whether the derived class has been derived or

If the derived class has been derived then a derived class object, can access **public** members of base class.

An object of a derived class (however derived) cannot access **private** or **protected** members of base class.

private members of base class cannot be accessed by derived class member functions or objects of derived class.

In **public** inheritance the **protected** members of the base class become **public** for the functions outside the derived class.

There is no difference between **private** and **protected** inheritance.

In **private** inheritance part of the base class interface can be made available to the functions outside the derived class.

The size of a derived class object is equal to the sum of sizes of data members in base class and the derived class.

Creating a derived class from a base class requires fundamental changes to the base class.

If a base class contains a member function and a derived class does not contain a function with this name, an object of the derived class cannot access **func()**

If no constructors are specified for a derived class, objects of the derived class will use the constructors in the base class.

If a base class and a derived class each include a member function with the same name, the member function of the derived class will be called by an object of the derived class

A class **D** can be derived from a class which is derived from a class which is derived from a class

It is illegal to make objects of one class members of another class.

[B] Answer the following:

Implement a **String** class containing the following functions:

- Overloaded + operator function to carry out the concatenation of strings.
- Overloaded = (assignment) operator function to carry out string copy.

- Overloaded `+=` operator function.
- Function to display the length of a string.
- Function to display the size of a string.
- Function `toLowerCase()` to convert upper case letters to lower case.
- Function `toUpperCase()` to convert lower case letters to upper case.

Suppose there is a base class **B** and a derived class **D** derived from **B** has two **public** member functions **b1()** and whereas **D** has two member functions **d1()** and Write these classes for the following different situations:

- **b1()** should be accessible in **b2()** should not be.
- Neither nor **b2()** should be accessible in
- Both **b1()** and **b2()** should be accessible in

If a class **D** is derived from two base classes **B1** and then write these classes each containing a zero-argument constructor. Ensure that while building an object of type **D** firstly the constructor of **B2** should get called followed by that of **B1**. Also provide a destructor in each class. In what order would these destructors get called?

Assume a class **D** that is **private** ly derived from class Which of the following can an object of class **D** located in **main()** access?

- **public** members of **D**
- **protected** members of **D**
- **private** members of **D**
- **public** members of **B**
- **protected** members of **B**
- **private** members of **B**

KanNotes

C++ facilitates code reuse at 2 levels: a) Source code level b)
Object code level

Source code level reuse is done using Template functions and
Template classes

Templates let us write generalized functions / classes and the
compiler creates specific functions / classes from it

For creating specialized functions / classes source code has to
be available

Object code level reuse is done using Containership and
Inheritance

Containership should be used when the two classes have a
“has a” relationship

Inheritance should be used when the two classes have a “like
a” relationship

Containership and Inheritance can be implemented even if source code is not available

Inheritance terminology: base - derived, parent - child

Protected members are available in the inheritance chain

Derived class object contains all base class data

Derived class object may not be able to access all base class data

Construction of an object always proceeds from base towards derived

Base class constructor can be called using `baseclassname()`

Inheritance facilitates:

Inheritance of existing feature: To implement this just establish inheritance relationship

Suppressing an existing feature: Hide base class implementation by defining same function in derived class

Extending an existing feature: call base class function from derived class by using Baseclassname::Baseclassfunction();

There are 3 types of inheritance: 1) Simple 2) Multi-level 3) Multiple

In multiple inheritance a class is derived from 2 or more than 2 base classes

In multiple inheritance order of classes in the derived class declaration dictates the order in which the constructors of base classes are called

Polymorphism

After classes & objects and inheritance, polymorphism is the third important feature of C++. If you stop at inheritance, you will be missing out on the greatest part of the language, namely polymorphism. This chapter explains the polymorphism concept in detail.

Virtual Functions

Pure Virtual Functions

Abstract Class

Function Binding

Virtual Functions under the Hood

Why Use virtual Functions?

Object Slicing

Virtual Destructors

Calling Virtual Functions from Ctors / Destructors

Virtual Base Classes

Exercise

KanNotes

Programmers who switch over from C to C++ seem to do so in three steps. In the first step they start using C++ simply as “better C”. During this stage they start using function prototypes, scope resolution operator, const, references, and a few more small concepts. These don't have much to do with object oriented programming.

Once comfortable with the non-object-oriented features of the language the second step is to use C++ as a “object-based” programming language. This means that they start appreciating the benefits of grouping data together with the functions that act upon it, the value of constructors and destructors, and perhaps some simple inheritance.

Many programmers carry a wrong impression that since they have started using classes, objects and inheritance they have graduated to the object-oriented world. Though on the face of it everything may appear nice, neat and clean don't get fooled. The most important piece of C++ is polymorphism.

In C++ polymorphism is encountered in two primary forms:

One thing existing in several different forms

One action leads to different activities

We have already experienced the first form when we learnt overloaded functions and overloaded operators.

In the second form, the action is ‘calling a function’ and activity that results from it is, different functions getting called in a class hierarchy. This form of polymorphism is implemented using **virtual** functions.

Virtual Functions

Suppose we have two classes **Circle** and both inherited from a class called **Shape**. Each class contains a **draw()** function to draw the relevant shape on the screen. If we are to draw a picture containing numerous circles and rectangles, we can do so through a program given below.

```
#include
using namespace std;

class Shape
{
public:
    virtual void draw()
    {
        cout << "In Shape class" << endl;
    }
};

class Circle: public Shape
{
public:
    void draw()
    {
```

```

cout << "In Circle class" << endl;
}
};

class Rectangle: public Shape
{
public:
void draw()
{

cout << "In Rectangle class" << endl;
}
};

int main()
{
Circle c1, c2, c3;
Rectangle r1, r2, r3;
int i;

Shape *p[] = {&c1, &c2, &r1, &r2, &r3, &c3};
for (i = 0; i <= 5; i++)
p[i]->draw();
return 0;
}

```

Here is the output of the program...

```
In Circle class
In Circle class
In Rectangle class
In Rectangle class
In Rectangle class
In Circle class
```

Note that in this program **Shape** class as well as classes derived from it— **Circle** and **Rectangle** contain a method called `draw()`. Instead of drawing a shape, these functions merely display a message.

In **main()** we have created an array of **Shape** pointers and stored in it addresses of **Circle** and **Rectangle** objects. Though there is a mismatch in the type on left hand side and right hand side `* / Rectangle` this is permitted because **Shape** is base class and **Circle** and **Rectangle** are classes derived from it. When we store address of a derived class object in pointer to base class object, it is known as an upcasted pointer. Thus, **p[]** is an array of upcasted pointers.

When it is time to draw the picture we can simply run the loop,

```
for (i = 0; i <= 5; i++)
p[i]->draw();
```

When **p[i]** contains address of the **Circle** object it calls the **Circle::draw()** function. Similarly, when it contains the address of the **Rectangle** object it would call the **Rectangle::draw()** function. Thus through the same action, different activities result. In short, polymorphism at work.

For this polymorphic approach to work, several conditions must be met. These are:

The classes **Circle** and **Rectangle** must be derived from the same base class,

Circle and **Rectangle** class must contain a function called **draw()**.

The **Shape** class's **draw()** function must be declared

The call to **draw()** must happen through a pointer.

Let us consider two possibilities.

If **draw()** is not marked as **p[i]->draw()** would result in a call to

If `p[]` contains addresses of **Shape** objects, `p[i]->draw()` would result in a call to

Pure Virtual Functions

We can add another refinement to the virtual function declared in the **Shape** class of the last program. Since the function **draw()** in the base class never gets executed, we can easily do away with the body of this virtual function and add a notation =o in the function declaration, as shown below:

```
class base
{
public:
    virtual void draw() = 0;
};
```

The **draw()** function is now known as a **pure** virtual function. The = sign here has got nothing to do with assignment; the value o is not assigned to anything. It is used to simply tell the compiler that a function will be pure, i.e. it will not have any body.

If we can remove the body of the virtual function can we not remove the function altogether. That won't work. Without a function **draw()** in the base class, statements like

```
p[i]->draw();
```

would be invalid.

Abstract Class

Saying ‘draw a shape’ is not so meaningful, as it doesn't convey which type of shape is to be drawn. So we should somehow prevent a **Shape** object from being created. If a **Shape** object cannot be created, then there is no question of calling **draw()** using it. This purpose is served by a pure virtual function.

If a class that contains a pure virtual function then we cannot create an object from it. Such a class is known as an abstract class.

Whenever a pure virtual function is placed in the base class, you must override it in all the derived classes from which you wish to create objects. If you don't do this in a derived class then the derived class becomes an abstract class.

Note that an abstract class may contain some instance functions, some virtual functions and some pure virtual functions.

Function Binding

The term binding refers to the connection between a function call and the actual code executed as a result of the call. Binding essentially decides which function to call. If this decision can be taken at compile-time, it is called **static** or **early** binding. If this is determined i.e. during execution of the program then it is called **dynamic** or **late** binding.

Suppose there is a global function by name and it is called from some place in the program. As there is only one version of compiler has no confusion in binding this call to the definition of Same is true about an instance function being called using an object or a pointer to an object of which the function is a member.

How would an overloaded global function **abs()** be bound? In this case based on the type of argument used in the call, compiler can decide which version of **abs()** should be called. Thus in this case too, static binding would take place. Same would be the case if **abs()** is overloaded in a class and is being called using an object of that class or a pointer to that class.

Let us now see when a function has to be late bound.
Suppose we take the same example of base class called **Shape** and derived classes called **Circle** and Each of them has a **draw()** function and the base class version has been marked as Consider the following calls:

```
Shape s;  
Circle c;  
Rectangle r;  
s.draw(); // calls Shape::draw()  
c.draw(); // calls Circle::draw()
```

```
r.draw(); // calls Rectangle::draw()
```

Each of the calls are early bound as the calls are being made using specific object's. So there is no confusion as to which object's function should be called.

However, the following calls would be late bound:

```
Shape s;  
Circle c;  
Rectangle r;  
fun (&s);  
fun (&c);
```

```
fun (&r);
// some code

// function definition
void fun (Shape *p)
{
p->draw(); // late bound
}
```

Here all calls to **draw()** are late bound since the calls are being made using a pointer. Why can't these calls be bound at compile-time? That's because contents of **p** are not known at compile time. During execution, based on the contents of **p** the call to **draw()** is bound. For example in the first call to **fun()**, **p** contains address of **Shape** object, hence **p->draw()** calls In the second call **p** contains address of **Circle** object, hence **p->draw()** calls Lastly, in the third call **p** contains address of **Rectangle** object, hence **p->draw()** calls

You must have observed that in the first call the pointer is not upcasted, whereas, in the next two calls it is upcasted. Thus, we can conclude that the call would be late bound if the call is made using pointer, irrespective of whether it is upcasted or not.

Thus for late binding to take place following conditions are necessary:

There must be inheritance at work.

A function name by same prototype must be present in base class as well as derived class.

The base class function must be marked as virtual.

The call to the function must be made using a base class pointer (upcasted or not).

The keyword **virtual** tells the compiler it should automatically install all the mechanisms necessary to perform late binding. To understand these mechanisms we need to dig deeper. We propose to do that in the next section.

Virtual Functions under the Hood

Using virtual functions is only part of the story. Knowing how compiler implements them completes the other part. We have seen the first part.

Before we take up the second let us consider the following simple program.

```
#include  
using namespace std;  
  
class Sample  
{  
private:  
int i;  
public:  
virtual void display()  
{  
cout << endl << "In sample class";  
}  
};  
class Trial
```

```
{  
public:  
virtual void display()  
{  
cout << endl << "In example class";  
}  
};  
int main()  
{  
Sample s;
```

```
Trial t;  
cout << sizeof (s) << endl << sizeof (t);  
return 0;  
}
```

Here is the output of the program...

```
8  
4
```

To say the least, you would find the output surprising. We had only an **int** in the object still the size is being reported as 8 bytes. Without the the size of **t** is being reported only as 4 bytes. Let us try to find why this so happens.

When the class has **virtual** function(s), the size of any object of this class is the size of its data members plus the size of a pointer. This pointer is inserted in the object if you have one or more virtual functions in the class. This pointer is called VPTR, or Virtual Table (VTABLE) pointer.

To accomplish late binding, the compiler creates a table called VTABLE for each class that contains virtual functions and for the classes derived from it. The compiler places the addresses of the virtual functions for that particular class in the VTABLE. If you don't redefine a function that was declared virtual in the base class, the compiler uses the address of the base-class version in the derived class's VTABLE.

Thus a VPTR is on a per object basis, whereas a VTABLE is on a per class basis. Moreover, VPTR of an object always points to the VTABLE of the class from which the object is created. The VPTR is set up in the constructor.

When you make a virtual function call through a base-class pointer the compiler quietly inserts code to fetch the VPTR and look up the function address in the VTABLE, thus calling the right function and causing late binding to take place.

Suppose a pointer to the base class object contains address of the derived class object and you call a virtual function

using this pointer. Now something special happens. Instead of performing a typical function call, which is simply an assembly language CALL to a particular address, the compiler generates different code to perform the function call. The compiler starts with the contents of the base-class pointer. These contents are address of the derived class object. Using this address the VPTR of the derived class object is fetched. Using VPTR the VTABLE of the derived class is accessed. From this table the address of the function being called is extracted. Lastly using this address the function of the derived class is called.

All of this—setting up the VTABLE for each class, initializing the VPTR, inserting the code for the virtual function call—happens automatically, so you don't have to worry about it.

I am sure what we said here would seem pretty abstract unless we see it working in a program. So let us write one. [Figure 7-1](#) shows the hierarchy of classes that we propose to implement in the program.

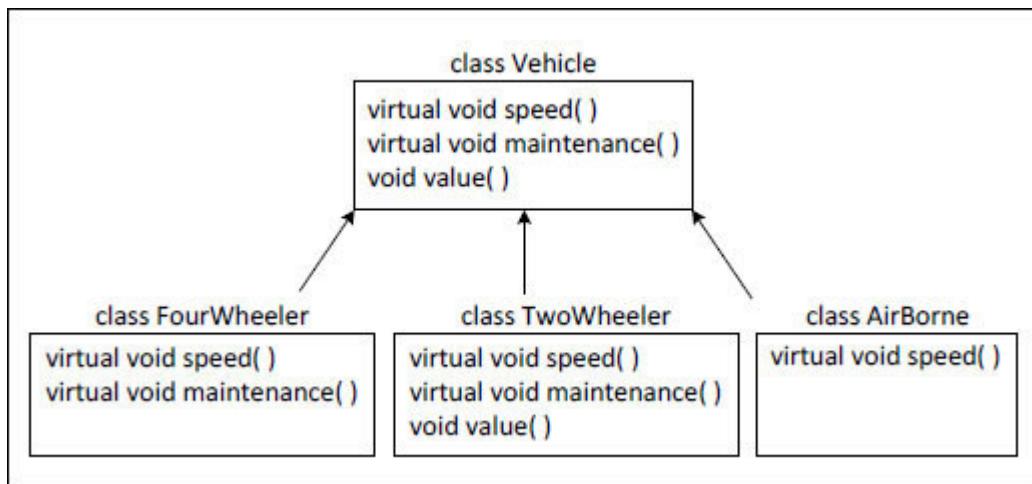


Figure 7-1. Class hierarchy.

Given below is the program that implements this class hierarchy.

```

#include
using namespace std;

class Vehicle
{
public:
    virtual void speed()
    {
        cout << endl << "In speed of Vehicle";
    }
    virtual void maintenance()
    {
  
```

```
cout << endl << "In maintenance of Vehicle" << endl;
}
void value()
{
cout << endl << "In value of Vehicle";
}
};

class FourWheeler: public Vehicle
{
public:
void speed()
{
cout << endl << "In speed of FourWheeler";
}
void maintenance()

{
cout << endl << "In maintenance of FourWheeler";
}
};

class TwoWheeler: public Vehicle
{
public:
void speed()
{
cout << endl << "In speed of TwoWheeler";
}
void maintenance()
```

```
{  
cout << endl << "In maintenance of TwoWheeler";  
}  
void value()  
{  
cout << endl << "In value of TwoWheeler";  
}  
};  
class AirBorne: public Vehicle  
{  
public:  
void speed()  
{  
cout << endl << "In speed of AirBorne";  
}  
};
```

```
int main()
```

```
{  
Vehicle *ptr1;  
Vehicle v;  
  
ptr1 = &v;  
ptr1 -> speed();  
ptr1 -> maintenance();  
ptr1 -> value();
```

```
Vehicle *ptr2, *ptr3, *ptr4;  
FourWheeler maruti;  
TwoWheeler bajaj;  
AirBorne jumbo;
```

```
ptr2 = &maruti;  
ptr3 = &bajaj;  
ptr4 = &jumbo;
```

```
ptr2 -> speed();  
ptr2 -> maintenance();
```

```
ptr3 -> speed();  
ptr3 -> maintenance();
```

```
ptr4 -> speed();  
ptr4 -> maintenance();
```

```
ptr2 -> value();  
ptr3 -> value();
```

```
Vehicle w;  
w.speed();
```

```
FourWheeler f;
```

```
f.speed();
```

```
AirBorne a;  
a.maintenance();  
return o;
```

```
}
```

Here is the output of the program...

In speed of Vehicle

In maintenance of Vehicle

In value of Vehicle

In speed of FourWheeler

In maintenance of FourWheeler

In speed of TwoWheeler

In maintenance of TwoWheeler

In speed of AirBorne

In maintenance of Vehicle

In value of Vehicle

In value of Vehicle

In speed of Vehicle

In speed of FourWheeler

In maintenance of Vehicle

As we saw earlier, a VTABLE is created for every class that contains virtual functions and for the classes derived from it. It means in our program a VTABLE would be built for all the four classes: **FourWheeler** and Each of these VTABLEs would contain addresses of the virtual functions. For all the objects built from these classes the compiler would automatically insert a VPTR which would point to the class's VTABLE. This is shown in the [Figure](#)

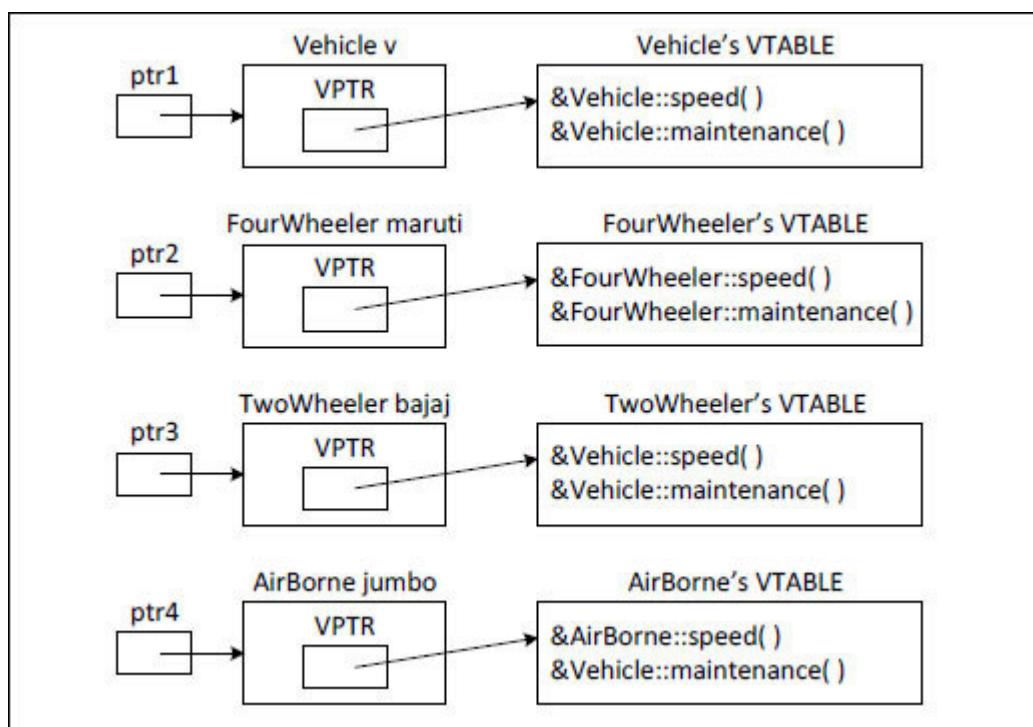


Figure 7-2. VPTRs and VTABLEs for class hierarchy.

Note that the class **AirBorne** doesn't contain the definition of the **maintenance()** function. Hence its VTABLE contains the address of the base class's **maintenance()** function.

Let us now understand the working of the program a step at a time. To begin with, we have stored the address of the base class object in a pointer to the base class through the statements

```
Vehicle *ptr1;  
Vehicle v;  
ptr1 = &v;
```

Next, we have called the member functions of the base class through the statements

```
ptr1 -> speed();  
ptr1 -> maintenance();  
ptr1 -> value();
```

Though these calls appear similar, behind the screen they are built differently. This is because the first two functions have been declared as **virtual** in the base class, whereas the third has not been. Since **value()** is not irrespective of the whether the base class object's or derived class object's address is

stored in always the base class's **value()** function would get called.

Since **speed()** has been defined **virtual** in the base class, which of its implementation (of **FourWheeler** or **TwoWheeler**) would be called depends upon whose address is stored in **ptr**. In our case this turns out to be **Vehicle** object's address. So while calling **speed()** firstly the VPTR is retrieved from the **Vehicle** object. Using this pointer the VTABLE of the **Vehicle** class is accessed. From this VTABLE the address of **speed()** (of **Vehicle**) is retrieved. Using this address **Vehicle::speed()** is ultimately called. Exactly same argument applies to calling **maintenance()**. Because the fetching of the **vptr** and the determination of the actual function address occurs at run-time, you get the desired late binding.

The next case is more interesting. Consider the statements

```
Vehicle *ptr2;  
FourWheeler maruti;  
ptr2 = &maruti;  
ptr2 -> speed();  
ptr2 -> maintenance();
```

Here we have stored the address of the derived class object in a pointer to the base class object. Now when we call **speed()**

VPTR of the object **maruti** would be used to access the VTABLE of the **FourWheeler** class. From this VTABLE the address of **FourWheeler::speed()** would be retrieved. Using this address the function **FourWheeler::speed()** would then get called.

If you think on these lines, you can understand the subsequent calls made using **ptr3** and

Now consider the calls

```
ptr2 -> value();  
ptr3 -> value();
```

Since **value()** has not been defined as **virtual** in the base class, VTABLEs are not involved in generating a call to In our program we have not overridden **value()** in the derived class. Had we done so, still it is the base class implementation that would have been called.

The next three calls are interesting.

```
Vehicle w;  
w.speed();
```

```
FourWheeler f;
```

```
f.speed();
```

```
AirBorne a;  
a.maintenance();
```

Here VPTR or VTABLEs are not involved at all. While calling **f.speed()** there is no ambiguity. As the compiler has an object (rather than its address) it knows the exact type and therefore it will not use late binding for any function calls. In general, for efficiency's sake, most compilers will perform early binding when they are making a call to a virtual function for an object because they know the exact type.

Now that we are through with the program let us look at a few more subtle issues related to virtual functions:

Because of the vital role played by VPTR while calling virtual functions it is critical that VPTR always points to the proper VTABLE. A call to a virtual function should not be made before VPTR is properly initialized. So the best place where this initialization can take place is the constructor. That is why the compiler always adds a zero-argument constructor to our class when we don't define one—to initiate the VPTR, if necessary.

How does the compiler manage to obtain the value of VPTR from the object. All objects have their VPTR in the same place (often at the beginning of the object), so the compiler can pick it out of the object easily.

Once a function is declared as **virtual** in the base class it is treated as **virtual** in the derived class, even though it has not been explicitly marked as **virtual** in the derived class.

Why use Virtual Functions?

That's an important question. If the technique of virtual functions is so good, and if it makes the 'right' function call all the time, why is it an option? Why do we even need to know about it?

The answer is simple. The mechanism of virtual functions is not very efficient. As compared to a simple CALL to an absolute address, there are more sophisticated assembly instructions required to set up the virtual function call. This requires both code space and execution time. Some object-oriented languages like Java use late binding for all function calls. Thus it is no longer an option, and the user doesn't have to know about it. However, C++ comes from the C heritage, where efficiency is critical. C++ is supposed to make C programmers more efficient. Had all function calls in C++ been implemented through late binding the efficiency would have suffered heavily.

Hence the virtual function is an option, and by default the language uses the non-virtual virtual mechanism which is of course faster. In short, if you don't use it, you don't pay for it.

While designing your classes, however, you shouldn't worry about loss of efficiency due to use of virtual functions. If you're going to use polymorphism, use virtual functions everywhere. Because by using virtual functions there are usually much bigger gains to be had in other areas.

Object Slicing

What do virtual functions achieve? They ensure that the code that manipulates objects of a base type can without change manipulate derived-type objects as well. Virtual functions should however always be called using either a pointer or a reference. If we try to do so using an object a phenomenon called **object slicing** takes place. The following program throws more light on this effect.

```
#include  
using namespace std;
```

```
class base  
{  
private:  
int i;  
public:  
base (int ii)  
{  
i = ii;  
}  
virtual void fun1()  
{
```

```
cout << endl << i << endl;
}
};

class derived: public base
{
private:
int j;
public:

derived (int ii, int jj): base (ii)
{
j = jj;
}
void fun1()
{
base::fun1();
cout << endl << j << endl;
}
};

int main()
{
base b (10);
derived d (15, 20);

base *ptr1 = &b;
ptr1->fun1();
```

```
base *ptr2 = &d;  
ptr2->fun1();
```

```
base &ref1 = b;  
ref1.fun1();
```

```
base &ref2 = d;  
ref2.fun1();
```

```
b = d; // object sliced  
b.fun1();  
return o;  
}
```

I am sure you would understand the calls made to **fun1()** using pointers and references. The problem occurs during the last assignment. When we use an object instead of a pointer or reference as the recipient of the upcast, the object is **sliced** until all that remains is the subobject that corresponds to the recipient. That is, if an object of a derived class is assigned to a base class object, the compiler accepts it, but it copies only the base portion of the object. It slices off the derived portion of the object. Hence when we make the call **b.fun1()** only the member function in the base class gets called.

Object slicing actually removes part of the object rather than simply changing the meaning of an address as when using a pointer or reference. Because of this, upcasting into an object is not often done: in fact, it's usually something to watch out for and prevent. You can explicitly prevent object slicing by putting pure virtual functions in the base class: this will cause a compile-time error when we try to create a base class object.

Virtual Destructors

The way the job of the constructor is to put together an object piece-by- piece, the job of the destructor is to break it part by part. While building as well as while destroying, each constructor and destructor in the class hierarchy should get called. The difference is only in the order of calling.

Construction starts by first calling the base constructor, then the more derived constructor and so on till the last class in the hierarchy chain. During destruction an exactly reverse order is followed. That is, the destructor starts at the most-derived class and works its way up to the base class. This is not only logical, but also safe. Let us see how.

A constructor of a derived class can access any **public** and **protected** member of the base class. Hence by the time they are accessed they must be properly set up. This can be ensured by calling the base class constructor before the derived class constructor.

Similarly, while destroying an object, the current destructor always knows that the base-class members are alive and active. It should not so happen that the base class members get destroyed through the base class destructor and then the

derived class destructor tries to access them. This can be ensured by first calling the derived class destructor followed by the base class destructor. Thus, the destructor can perform its own cleanup, then call the base class destructor, which will perform its own cleanup. This it can do because it knows what it is derived from, but not what is derived from it.

You should keep in mind that constructors and destructors are the only place where this hierarchy of calls must happen. In all other functions, only that function will be called, whether it's virtual or not. The only way for base-class version of the same function to be called in ordinary functions or not) is if you explicitly call that function.

Consider a situation where a derived class object is created using The address of this object can be assigned to a pointer to a base class object. Now if we **delete** the pointer, since the pointer is a base class pointer this would result in a call to the base class destructor. Ideally, firstly the derived class destructor should be called followed by the base class destructor. This can be ensured by using a **virtual** destructor in the base class. The following program shows how this can be implemented.

```
#include  
using namespace std;
```

```
class Base
{
public:
    Base()
    {
        cout << "In Base class constructor" << endl;
    }
    virtual ~Base()
    {
        cout << "In Base class destructor" << endl;
    }
};

class Derived: public Base
{

public:
    Derived()
    {
        cout << "In Derived class constructor" << endl;
    }
    ~Derived()
    {
        cout << "In Derived class destructor" << endl;
    }
};

int main()
{
```

```
Base *b;  
b = new Derived;  
delete b;  
return o;  
}
```

Here is the output of the program...

```
In Base class constructor  
In Derived class constructor  
In Derived class destructor  
In Base class destructor
```

Even though the destructor, like the constructor, is an exceptional function, it is possible for the destructor to be **virtual** because the object already knows what type it is (whereas, it doesn't during construction). Once an object has been constructed, its VPTR is initialized, so virtual function calls can take place.

The way we can create a pure virtual function, we can create a pure virtual destructor as well. However, we must provide a function body to the pure virtual destructor because (unlike ordinary functions) all destructors in a class hierarchy are always called. Thus, the body for the pure virtual destructor ends up being called.

By declaring the virtual destructor as pure in the base class, we are in effect forcing the derived class to redefine the destructor. The base class body of the destructor is still called as part of destruction.

As a guideline, any time you have a virtual function in a class, you should immediately add a **virtual** destructor (even if it does nothing). This way, you can prevent any surprises later.

CALLING VIRTUAL FUNCTIONS FROM CONSTRUCTORS / DESTRUCTORS

When we call a virtual function from inside an ordinary member function the late-binding mechanism is used for the call. This is not true with constructors or destructors. From inside a constructor/destructor, only the local version of the member function is called; the virtual mechanism is ignored. This can be confirmed from the following program.

Virtual Base Classes

Let us look at one more subtlety of virtual functions. Consider the situation where there is one parent class called **Base** and two classes derived from it, **Derived1** and **Derived2**. Suppose we derive a class **Derived3** from **Derived1** and Now suppose a member function of **Derived3** class wants to access data or functions in the **Base** class. Since **Derived1** and **Derived2** are derived from **Base** each inherits a copy of This copy is referred to as a Each subobject contains its own copy of data. This is shown in [Figure](#)

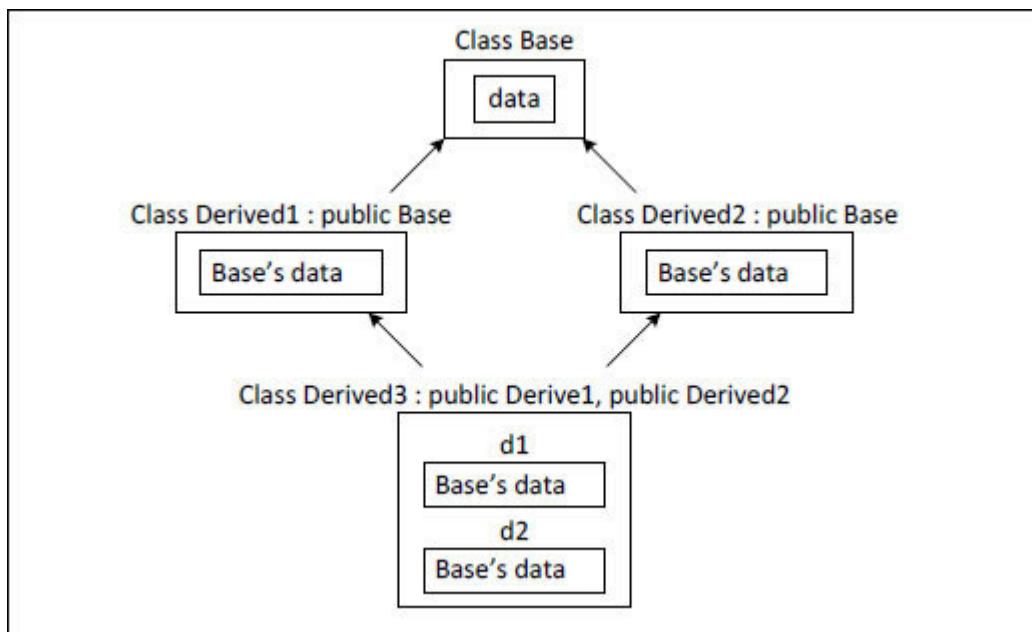


Figure 7-3. Multiple subobjects during multiple inheritance.

Now when **Derived3** refers to the data in the **Base** class, which of the two copies will it access? This is an ambiguous situation for the compiler, hence it reports an error. To get rid of this ambiguity, we should make **Derived1** and **Derived2** as **virtual** base classes as shown in the following program.

```
#include
using namespace std;

class Base
{
protected:
int data;
public:
Base()
{
data = 500;
}
};

class Derived1: virtual public Base
{
};

class Derived2: virtual public Base
{
};

class Derived3: public Derived1, public Derived2
```

```

class Derived3: public Derived1, public Derived2
{
public:
int getData()
{
return data;
}
};

int main()
{
Derived3 ch;
int a;
a = ch.getData();

cout << a << endl;
return 0;
}

```

Using the keyword **virtual** in the two base classes causes them to share the single subobject of the **Base** class. Since there is only one subobject there is no ambiguity when it is referred in the **Derived3** class. Here **Derived1** and **Derived2** are known as **virtual base**

Suppose there is a function **fun()** in **Base** class and we access it using a **Derived3** object. In such a case compiler reports an

error since **Derived3** would inherit two copies of one via **Derived1** and another via Hence when we attempt to call **fun()** the compiler would not know whether we intend to call the copy of **Derived1** or that of Again the error can be overcome by using declaring **Derived1** and **Derived2** as virtual base classes.

The inheritance hierarchy used in this program is often called diamond pattern (refer [Figure](#)

Exercise

[A] State True or False:

Virtual functions permit calling of derived class functions using a base class pointer.

There is one VPTR per VTABLE and one VPTR per object.

VPTR always points to the VTABLE of the class.

Virtual functions permit functions from different classes to be executed through the same function call.

Pure virtual functions can never have a body.

We can never build an object from a class containing a pure virtual function.

A class containing a pure virtual function is called an abstract base class.

Virtual function calls work faster than normal function calls.

In a class hierarchy of several levels if we want a function at any level to be called through a base class pointer then the function must be declared as **virtual** in the base class.

Virtual functions can be safely invoked using objects.

The behavior of **virtual** functions is same irrespective of whether we invoke them through pointers or references.

While building an object it doesn't matter whether the base class constructor is called first or the derived class constructor is called first.

While destroying an object firstly the derived class destructor should be called followed by the base class destructor.

A **virtual** destructor ensures a proper calling order for the destructors in the class hierarchy.

A pure **virtual** destructor can have a function body.

[B] Answer the following:

What are virtual base classes? When should they be used?

Write a program that contains a class derived from **Base** class should have a **virtual** function **fun()** and it should be overridden in Try to call **fun()** from the constructor of the derived class and watch the results.

KanNotes

Polymorphism has two meanings:

- 1) One thing existing in several forms
- 2) One action results into different activities

Virtual functions help implement the second type of polymorphism

Binding means deciding which function to call

If binding is done at the time of compilation it is called Early Binding

If binding is done at the time of execution it is called Late Binding

C++ - Does Early Binding when possible and Late Binding when Early Binding is not possible

Early Binding is also known as Static Binding or Compile time Binding

Late Binding is also known as Dynamic Binding or Runtime Binding

For Late Binding the function being called must be present in base class as well as derived class and the call must be made using a pointer

If call is made using an object, then it is always early bound

Upcasted pointer - a base class pointer containing address of derived class object

If call is made using a pointer (upcasted or not) it is late bound, if function is virtual

To prevent an object from getting created from a class it should contain at least one pure virtual function

A class from which an object cannot be created is called an abstract class

If a class contains a virtual function a VTABLE is created for it. All objects of this class will have VPTR in them

If base class contains a virtual function and the class derived from it contain a function having same prototype, then the derived class function is treated as virtual even though it is not marked virtual explicitly

If base class contains a virtual function and the class derived from it doesn't contain a function having same prototype, then the derived class's VTABLE contains the address of the base class virtual function

A virtual destructor ensures a proper calling order for the destructors in the class hierarchy

In a diamond pattern inheritance creation of multiple subobjects can be prevented by marking intermediate classes as virtual

Input / Output in C++

There is not much use of writing a program that spends all its time telling a secret to itself. Almost all programs have to perform Input/Output in some form or the other. This chapter explores this important aspect of C++ programming.

Expectations from an I/O System

C++ Streams Solution

Ready-made Stream Objects

The iostream Library

The istream Class

The ostream Class

Printing Unicode Characters

The *iostream* Class

Stream Manipulators

User-defined Manipulators

User-defined Manipulators with Arguments

File I/O with Streams

Character I/O

Opening File

Reading Data

Detecting End of File

Closing File

A Filecopy Program

Line I/O

Record I/O

Random Access

File Opening Modes

String Streams

Using *istrstream*

Object I/O

Serialization

Error Handling During I/O

Interaction with File System

Exercise

KanNotes

Computing languages have been around for more than 50 years. So when C++ came along riding on the back of OOP, programmers expected a mature input/output system that was easy, clean, safe and adaptable. We have already experienced these aspects of I/O while using **cout** and **cin**. Let me explain how.

While using **cout** and **cin** we are not required to remember format specifiers like %c, %d, %f etc., which are mandatory in **scanf()** and **printf()** in C programming. In that sense it is easy to use them.

cout and **cin** are adaptable because we can make them input or output objects of user-defined types.

If an I/O operation has not been defined for a particular type, compiler will generate an error when we try to use **cout** and **cin** with this type.

But **cout** and **cin** together form a very small part of all the facilities that C++ I/O system offers to programmers. This chapter proposes to explore these facilities with an aim to effectively carry out I/O needs of a program.

Expectations from an I/O System

Programmers typically have following expectations from an I/O system:

Communication with different sources and A C++ program should be able to carry out reading operations from input devices like keyboard, port, disk, etc., and perform writing operations to disk, printer, port, etc.

Capability to I/O varied A C++ program should be able to I/O byte, char, numbers of all kinds, strings, records and objects.

Multiple means of A C++ program should be able to carry out I/O in different modes like sequential and random.

Communication with file A C++ program should be able to interact with file system entities like files and directories and be able to access and manipulate paths, times, dates, access permissions, etc.

A robust error-reporting A C++ program should be able to effectively deal with errors that happen while performing I/O

operations.

Let us now see how C++ meets these expectations.

The C library for performing I/O was based on functions, whereas the one in C++ is based on classes. Both the languages do not have keywords to perform I/O. The C library for performing I/O is often known as **stdio** library, whereas, the C++ library is known as **iostream** library.

C++ Streams Solution

To meet the expectations of a mature I/O system C++ designer decided that all I/O should be performed using I/O Streams. A stream is a sequence of bytes that travel from source to destination over a communication path. A program can read data from a stream or write data to a stream. The streams are linked to physical devices by C++ I/O system. Most of the communication details are hidden from us by the I/O system and we are required to concentrate only on what we wish to read from where, and what we wish to write where. [Figure 8-1](#) should help you understand this concept better.

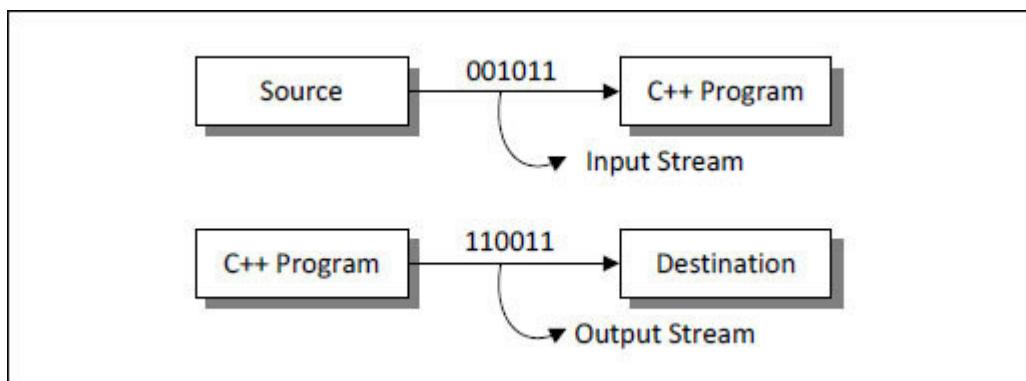


Figure 8-1. I/O streams.

Streams are implemented using classes in the **iostream** library. This abstraction of I/O operations using streams offers one important benefit—no matter from where we are reading or where we are writing, stream behaves similarly. For example, whether we are reading from a network connection or a disk file we call the same method. The implementation of the method is different for different devices. Thus, because of stream-based I/O the programmer doesn't have to worry about the specific details of the operating system and underlying devices while performing I/O as shown in [Figure](#). The differences in the devices and the OS are hidden away from us into different stream classes in the **iostream** library.

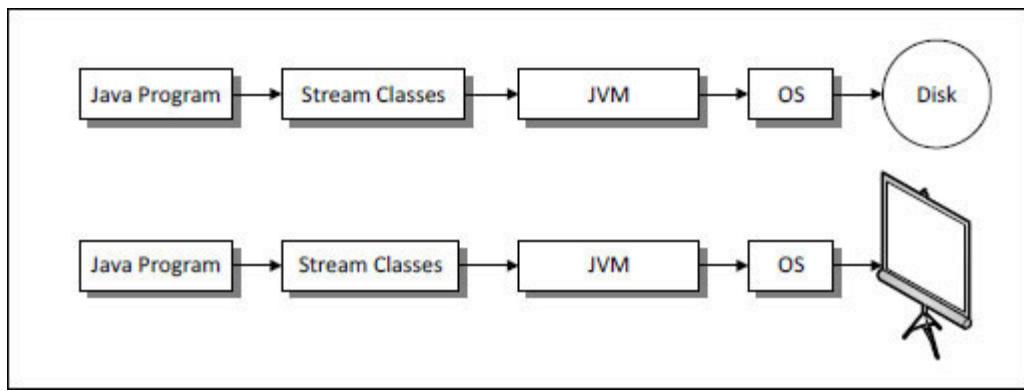


Figure 8-2. Stream based output to different devices.

The two fundamental operations that can be performed on a stream are Reading and Writing. Reading involves transfer of data from a stream into a data structure, such as an array of bytes. Writing consists of transfer of data from a data source

into a stream. Every stream may not support reading and writing.

Ready-made Stream Objects

As we saw, a **stream** is a general name given to the flow of data. Different streams are used to represent different kinds of data flow. For example, the standard input stream flows from the keyboard to the program. Likewise, standard output stream, standard error stream and standard log stream flows from program to screen.

In C++ a stream is represented by an object of a particular class. Hence for the streams mentioned above, **iostream** library provides ready-made objects **cerr** and **clog** respectively. Of these, **cin** is an object of **istream** class and **cout** and **clog** objects of **ostream** class. They all are used to perform character I/O for ASCII characters (1-byte). For Unicode characters (2-byte, also called wide-characters) their parallels **wcout** and **wcerr** and **wclog** are used. These are objects of **wistream** and **wostream** classes respectively.

You must have noticed that the objects **cerr** and **clog** all represent data flow from program to screen. Then what is the difference between the three? Well, **cout** is used to send output to screen, **cerr** is used to send errors to standard error device, and **clog** is used to send log messages to standard log device.

Usually, the standard error device and standard log device is screen.

Note that output sent using **cerr** is unbuffered, whereas the one sent using **clog** is buffered. This means that errors sent using **cerr** causes them to appear immediately—this is appropriate as the user comes to know about it promptly. As against this, output sent using **clog** object is buffered. This means that messages sent using **clog** could cause the messages to be held in a buffer. These messages are flushed to the screen once the buffer becomes full. Often the log messages are redirected to a file so that the screen is not cluttered with them.

The **iostream** Library

The **iostream** library provides many template classes for handling common I/O operations. Template classes are generic classes from which classes for specific types can be created. The partial hierarchy of classes in the **iostream** library is shown in [Figure](#)

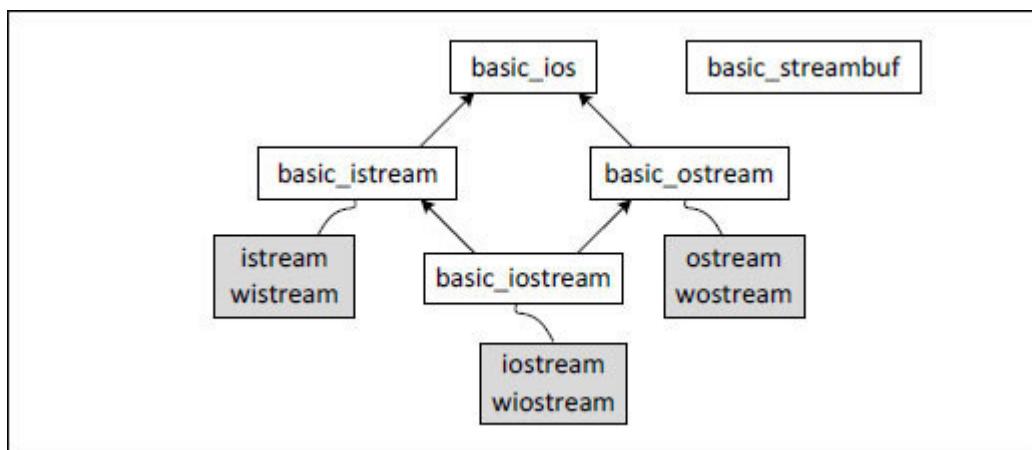


Figure 8-3. I/O classes hierarchy.

As seen from [Figure 8-3](#), the I/O specification class is at the root of the **iostream** class hierarchy. This class contains features that are common to all streams. These include flags for formatting the stream data, the error-status flags and the file operation modes. We will soon examine these in detail.

The **ios** class contains a pointer to the **basic_streambuf** class, which contains the actual memory buffer into which data is read or written, and the routines to handle its data. Usually, you don't need to worry about the **basic_streambuf** class, which is referenced by other classes. However, if required, you can access the buffer by calling the member functions of the **basic_streambuf** class.

The **basic_istream** is a template class. From this generic class, specific class **istream** and **wistream** are created for performing stream-input operations on chars and wide-chars respectively. Likewise, template class **basic_oiostream** and specific classes **ostream** and **wostream** support stream-output operations. Similarly, **basic_iostream**, **iostream** and **wiostream** support both stream-input and stream-output operations.

The **istream** Class

The **istream** class is derived from the **ios** class. It performs activities specific to input. One of the most commonly used member function of this class is the overloaded **>>** operator. It has been overloaded to extract values of all basic types. We can extract even a string using this operator as shown below

```
char str[100];
cin >> str;
```

The direction of **>>** gives you an idea of source and target. Here characters are flowing from input stream object **cin** into the string. However, in this case we have no control over the number of characters that would get extracted into the string. If the string overflows, it might be dangerous. Moreover, if we supply a multi-word string, **cin** extracts only the first word in it into. In such cases we can use the **get()** or **getline()** member function of **istream** class. The **get()** function comes in several forms. These are as under:

Function	Purpose
get (ch)	Extracts one character into ch
get (str, MAX)	Extracts up to MAX characters into str
get (str, DELIM)	Extracts characters into array str until specified delimiter (typically '\n'). Leaves delimiting character in stream
get (str, MAX, DELIM)	Extracts characters into str until MAX characters or the DELIM character. Leaves delimiting character in stream
getline (str, MAX, DELIM)	Extract characters into array str , until MAX characters or the DELIM character. Extracts delimiting character

Table 8-1. *Input functions in istream class.*

In addition to these, the **istream** class supports a few miscellaneous functions. These are shown in [Table](#)

Function	Purpose
putback (ch)	Inserts last character read, back into input stream
peek (ch)	Reads one character, leaves it in stream
num = gcount()	Returns number of character read by a (immediately preceding) call to get() , getline() , or read()
ignore (MAX, DELIM)	Extracts and discards up to MAX characters until (and including) the specified delimiter (typically '\n')

Table 8-2. *Miscellaneous functions in istream class.*

The following program puts these functions to work.

```
#include
using namespace std;

int main()
{
    char ch;
    cout << endl << "Enter a character:";
    cin.get (ch);
    cout << ch;
    cin.putback (ch);
    cin.get (ch);
    cout << endl << ch;
    int count = cin.gcount();

    cout << endl << "Characters extracted in last get() =" <<
    count;
    // stuff stream with a Z
    cin.putback ('Z');
    ch = cin.peek();
    cout << endl << ch;
    // Z is still in stream
    cin.get (ch);
```

```
cout << endl << ch << endl;
return o;
}
```

Lastly, **istream** class consists of functions that work specifically with files. These would be discussed when we do file I/O later in this chapter.

The ostream Class

The **ostream** class handles output or insertion activities. The most commonly used member function of this class is the overloaded << operator function. Two other useful member functions of this class are **put()** and **flush()**. The first one puts a character into the stream, whereas, the second flushes the buffer contents and inserts a newline.

Apart from standard data types strings can also be sent to console using << operator. The program given below shows a few typical usages of

```
#include
using namespace std;
int main()
{
    int ch = 90;
    cout << char (65) << endl;
    cout << char (ch) << endl;
    char str[] = "Be silent. Let performance speak!";
    char *p = "Be eloquent. Express yourself!";
    cout << str << endl;
    cout << p << endl;
```

```
cout << static_cast < void * > (str) << endl;
cout << static_cast < void * > (p) << endl;
return o;
}
```

This program produces the following output:

```
A
Z
Be silent. Let performance speak!
```

```
Be eloquent. Express yourself!
0032FBEC
oooC783o
```

Note that to print a character equivalent to 65 and 90, we have to cast the number into a `char *`. Also, to get the address of the string `str` or the address stored in pointer `p` we need to cast the `char *` into a `void *` using the static cast syntax. This type of typecasting is discussed in [Chapter](#)

PRINTING UNICODE CHARACTERS

Let us look at another interesting program. Suppose we wish to output characters in scripts like Devanagari or Cyrillic. For this we need to first obtain the Unicode values for characters

in these scripts. These are available at Unicode values for Cyrillic characters ?, ?, ?, ?, ? are hexadecimal 0411, 0414, 0416, 0419 and 041B. Similarly, Unicode values for characters ?, ?, ?, ? and ? are hexadecimal 0905, 0906, 0907, 0908 and 0909. The program given below shows how to use these values to print their Cyrillic and Hindi equivalents.

```
#include
#include
#include
using namespace std;

int main()
{
    _setmode (_fileno (stdout), _O_U16TEXT);
    wcout << "Following characters are in Cyrillic" << endl;
    wcout << L"\u0411\u0414\u0416\u0419\u041B" << endl;

    wcout << "Following characters are in Devanagari" << endl;
    wcout << L"\u0905\u0906\u0907\u0908\u0909" << endl;
    return 0;
}
```

On execution of this program we get the following output:

Following characters are in Cyrillic

БДЖЙЛ

Following characters are in Devanagari

अआइई

Now, a few points that you must note about this program.

The **_setmode()** function should be called to indicate we are sending 16-bit Unicode characters to the screen. The prototype of this function is declared in the file ‘fcntl.h’.

The macro **_O_U16TEXT** is defined in the file ‘io.h’.

To indicate that a string is a Unicode string, we need to prepend it with ‘L’. \x before each Unicode value indicates that it is a hexadecimal value.

Since we are printing 16-bit Unicode values we have to use **wcout** in place of

When you execute the program you may get the Cyrillic output alright, but instead of Devanagari characters, only boxes may appear. This is because the console window doesn't know how to print Devanagari characters.

When the same Unicode values were used to print characters in a Windows application, I got the following message box.



The **iostream** Class

The **iostream** class is derived from both **istream** and **ostream** by multiple inheritance. It acts only as a base class from which other classes can be derived. Other than the constructors and destructors, it doesn't contain any other member functions. As you can guess, classes derived from **iostream** can perform both input and output.

Stream Manipulators

C++ provides various stream manipulators that perform formatting tasks. The stream manipulators provide capabilities such as setting field widths, setting precision, flushing streams, inserting a newline into the output stream, skipping white space in the input stream, etc.

Manipulators come in two flavors—those that take argument and those that don't. Manipulators with no arguments are provided in ‘iostream’, whereas those that take arguments are provided in ‘iomanip’. [Table 8-3](#) gives a list of all manipulators along with their purpose.

Manipulator	Purpose
skipws / noskipws	Skip / Don't skip whitespace on input
dec / oct / hex	Convert to decimal, octal, hexadecimal
left / right	Left / Right align, pad on right/left
internal	Use padding between sign or base indicator and value
endl	Insert newline and flush output stream
showpos / noshowpos	Show/hide plus sign for positive values
uppercase / nouppercase	Show / Don't show uppercase A-F for hex values, and E for scientific values
showpoint / noshowpoint	Show / Don't show decimal point and trailing zeros for float values
scientific / fixed	Use scientific / fixed notation for printing float values
ends	Insert null char to terminate a string
flush	Flush the output stream
lock / unlock	Lock / Unlock file handle
setw (int n)	Change the field width for output to n
setfill (char n)	Change the fill character to n (default is a space)
setprecision (int n)	Set precision to n places after decimal point
setbase (base n)	Change base to n , where n is 8, 10 or 16
setiosflags (fmtflags n)	Set format flags specified by n . Setting remains in effect until next change
resetiosflags (fmtflags n)	Clear only the format flags specified by n . Setting remains in effect until next change

Table 8-3. Various stream manipulators.

The following program shows how to use these manipulators.

```
#include
#include

using namespace std;

int main()
{
    int i = 752;
    float a = 425;
    float b = 123.500328f;
    char str[] = "Dream. Then make it happen!";

    ios_base::fmtflags oldFlags;
    oldFlags = cout.flags();
    cout << "Current flags =" << cout.flags() << endl;

    cout << hex << i << endl;
    cout << showbase << hex << i << endl;
    cout << uppercase << showbase << hex << i << endl;
    cout << dec << i << endl;
    cout << internal << showpos << setw (10) << i << endl;
    cout << i << endl;
```

```

cout << setfill ('o');
cout << "Fill character:" << cout.fill() << endl;
cout << setw (40) << str << endl;
cout << left << setw (40) << str << endl;

cout.precision (6);
cout << "Precision:" << cout.precision() << endl;
cout << showpoint << showpos << a << endl;
cout << fixed << b << endl;
cout << scientific << b << endl;

cout << "Current flags =" << cout.flags() << endl;
cout.flags (oldFlags);
cout << "Current flags =" << cout.flags() << endl;

return o;

}

```

Here is the output of the program...

```

Current flags = 513
2fo
0x2fo
0X2Fo
752

```

+ 752

+752

Fill character:o

ooooooooooooooDream. Then make it happen!

Dream. Then make it happen!oooooooooooo

Precision: +6

+425.000

+123.500328

+1.235003E+002

Current flags = +4733

Current flags = 513

The output can be understood by looking up the purpose of the stream manipulators from [Table](#). Note that the manipulators like left, etc. stick around. It means once set, they remain effective for subsequent **cout** statements too.

Instead of resetting them one by one, it is a good idea to capture the default flag settings before we start manipulating the stream, then perform all manipulations and finally restore the captured settings.

The **flags()** member function of **ios_base** class returns the current format settings as a **fmtflags** data type. We have assigned these settings to **oldFlags** variable. Once we are done with the manipulations, we have restored the old settings by calling **flags()** and passing to it the settings preserved in The

initial settings value that `flags()` returns might differ across different systems.

User-defined Manipulators

Though the list of manipulators provided by the **iostream** library is quite impressive, at times you may want to create your own manipulators. Here I would demonstrate how to create both types of manipulators— those that do not take arguments and those that do. Let us begin with the first one.

To understand how to develop a zero-argument manipulator we need to understand the internal working of some existing manipulator, say **endl** is a function whose declaration in ‘iostream’ looks like this.

```
ostream& endl (ostream&);
```

Now consider the statement

```
cout << endl;
```

Since `<<` is an overloaded operator, internally this statement becomes,

```
cout.operator << (endl);
```

endl() being a function, its address gets passed to the overloaded operator function. The `<<` operator has been defined in ‘iostream’ as follows:

```
ostream & ostream::operator << (ostream & (*_f) (ostream &))
{
    return (*_f)(*this);
}
```

This indicates that when we pass the address of **endl()** to this function it collects it in a pointer to a function that receives an **ostream** reference and returns an **ostream** reference. If you observe carefully this matches the prototype of the **endl()** function. Since the `<<` operator function is called through the **cout** object, the **this** pointer contains the address of Hence ***this** yields the object. This object is then passed to the **endl()** function through the statement.

```
(*_f)(*this);
```

On getting called, all that the **endl()** function does is emit a ‘\n’ to the output stream.

Simple enough! Now we can proceed to develop our own manipulator called Here it is.

```
#include
using namespace std;

ostream & tab (ostream &o)
{
    return o << '\t';
}

int main()
{
    cout << "Don't";
    cout << tab << tab;
    cout << "panic" << endl;
    return 0;
}
```

This program works exactly on the similar lines as the previous one. The only difference is instead of calling the function here the **tab()** function gets called. Trace the flow of control in the program in a debugger. This would help you understand its working better.

User-defined Manipulators with Arguments

Let us create a manipulator called **roman** which receives an **unsigned long** as an argument and output its roman equivalent. To implement this manipulator we need to define a class called This class consists of a constructor and an overloaded << operator function. This function is implemented as a **friend** function. Here we would not discuss why the function has to be a **friend** function. If you can't wait and want to know friend functions this very moment you can jump to [Chapter](#) Others can take a look at the listing of the program that defines and uses the **roman** class.

```
#include
using namespace std;

class roman
{
private:
    unsigned long num;
public:
    roman (unsigned long n);
    friend ostream & operator << (ostream& o, roman& r);
};
```

```
roman :: roman (unsigned long n)
{
    num = n;
}
ostream & operator << (ostream& o, roman& r)
{
    struct key
    {

        char ch;
        int val;
    };
    key z[] = {
        {'m', 1000}, {'d', 500}, {'c', 100}, {'l', 50},
        {'x', 10}, {'v', 5}, {'i', 1}
    };
    int sz, k;

    sz = sizeof (z) / sizeof (z[0]);
    for (int i = 0; i < sz; i++)
    {
        k = r.num / z[i].val;
        for (int j = 1; j <= k; j++)
            o << z[i].ch;
        r.num = r.num % z[i].val;
    }
    return o;
}
```

```
int main()
{
long yr = 2019;
cout << roman (yr);
cout << endl << roman (1752) << endl;
return 0;
}
```

Here is the output of the program...

```
mmxviii
mdcclii
```

Observe the following statement carefully

```
cout << roman (yr);
```

Here **roman (yr)** creates a temporary object of the type **roman**. Naturally, while creating this temporary object the constructor gets called and the value passed to it gets set in the **private** variable. This temporary object and the **cout** object are then passed to the overloaded **operator << ()** function. Note that in case of a **friend** the call to the operator function **doesn't** get converted into the form

```
cout.operator << (roman (yr));
```

Within the operator function we have converted the value in **num** into its roman equivalent and outputted it using the reference of the **cout** object. In the end, we have returned the reference. This returning of reference is necessary if we are to use the manipulator in a cascaded **cout** like

```
cout << endl << roman (1752) << endl;
```

Thus we can implement a manipulator with an argument by developing a class for it. Does this mean that for every one-argument manipulator provided by the **iostream** library there is a separate class? Yes. However, instead of writing a separate class for each manipulator, the writers of the **iostream** library have put together a generic class hidden behind a complicated set of in the file ‘iomanip’. If you wish you can examine this file and try to understand this generic class.

File I/O with Streams

So far we have used `istream` and `ostream` objects (`cn` and `cout`) to perform console I/O operations. On similar lines we can do file I/O operations. This time we have to use objects of **`ifstream`** (for input), **`ofstream`** (for output) and **`fstream`** (for input and output) classes. As you must have guessed, these classes are specializations of template classes **`basic_ifstream`** (for file input), **`basic_ofstream`** (for file output) and **`basic_fstream`** (for file input and output). The hierarchy of these classes is shown in [Figure](#)

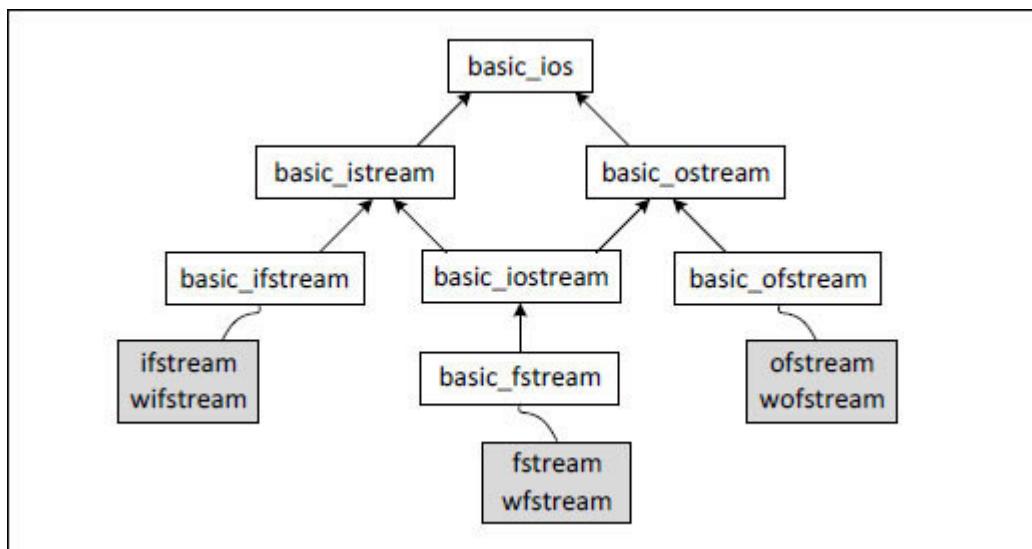


Figure 8-4. File I/O classes hierarchy.

Due to the inheritance relationship all member functions, operators and manipulators that belong to base class templates also can be applied to file streams.

When it comes to reading/writing data from/to a file there are essentially following three scenarios:

Reading/Writing a file character by character

Reading/Writing a file line by line

Reading/Writing a file record by record

Let us understand these scenarios using specific programming situations.

Character I/O

Let us now look at a simple program that reads a file character by character and displays it on the screen. Here is the program.

```
#include  
#include  
using namespace std;  
  
int main()  
{  
char ch;  
  
ifstream infile ("Sample.cpp");  
while (! infile.eof())  
{  
infile.get (ch);  
cout << ch;  
}  
infile.close();  
return 0;  
}
```

Though this is a small program, there are several new things here. Let us understand them one by one.

OPENING FILE

To begin with, we have defined an object called **infile** of type **ifstream** class through the statement

```
ifstream infile ("Sample.cpp");
```

This invokes the one argument constructor of the **ifstream** class. This constructor allocates resources and opens the file 'Sample.cpp'. But we didn't mention whether the file is to be opened for reading or writing. This is not necessary since the constructor uses the defaults. The prototype of the constructor looks like this.

```
ifstream (const char*, int = ios::in);
```

When we do not pass the second parameter to this constructor it uses the **ios::in** as the default file opening mode. Hence the file gets opened for reading.

Sometimes, we may not know the name of the file when the **ifstream** object is created. In such a case we may first create the object and then call **ifstream::open()** function to open the file. This procedure is shown below.

```
ifstream infile;  
infile.open ("Sample.cpp");
```

Once again we have not mentioned the file opening mode. The reason is same- **ios::in** has been binarily included (ORed) into the mode (second parameter).

READING DATA

Once the file is opened we have used the **ifstream::get()** function to read the file one character at a time. Each character read from the input file is displayed on the screen using the usual **cout** object.

The **ifstream** and **fstream** classes are declared in the header file 'fstream'. Hence we have include it explicitly.

DETECTING END OF FILE

Within the **while** loop we keep checking whether we are through with reading the entire contents of the file. For this

we have called the function. This function returns a value true if the end of file is reached, false otherwise.

We can get the same effect through a statement like

```
while (infile)
```

CLOSING FILE

We have specifically closed the file once reading is over, by calling the function. Closing the file disassociates it from the stream.

The program would work correctly even if we do not close the file once reading from the file is over. This is because on termination of the program the **infile** object would go out of scope. As a result, the destructor would get called and it would close the file.

A Filecopy Program

Let us now try to write a program that copies the contents of one file into another. To perform file copying we would read the source file character by character and keep writing every character read to the target file. The process would continue till the end of source file is not reached. Here is the program.

```
#include  
#include  
#include  
using namespace std;  
  
int main()  
{  
char ch;  
string source, target;  
cout << "Enter source file name:";  
cin >> source;  
cout << "Enter target file name:";  
cin >> target;  
ifstream infile (source);  
ofstream outfile (target);
```

```
while (infile)
{
    infile.get (ch);
    outfile.put (ch);
}
```

```
return o;
}
```

Here the characters are read using the **istream::get()** function and they are written to the target file using

Line I/O

If the file used for reading/writing is a text file then we can process it either character by character or line by line. The first operation should be used if we really wish to process each character that is read. For example, suppose we wish to read a character, encrypt it and then write the encrypted character to a target file. In this case character I/O makes sense.

However, if we are to process characters in bulk, as in a file-copy operation, then we should prefer line I/O. Each line in a text file can be treated as a string. The following program shows how the file-copy program can be improved by reading/writing a file line by line.

```
#include  
include  
#include  
using namespace std;  
int main()  
{  
    string source, target;  
    char str[80];
```

```
cout << "Enter source file name:";  
cin >> source;
```

```
cout << "Enter source file name:";  
cin >> target;
```

```
ifstream infile (source);  
ofstream outfile (target);
```

```
while (infile)  
{  
    infile.getline (str, 79, '\n');  
  
    outfile << str << endl;  
}
```

```
infile.close();  
return o;  
}
```

Here we have read the text from the file one line at a time using the **getline()** function. This function is a member of **istream** (from which **ifstream** is derived). Reading of a line ends either when 79 characters in the line have been read or an end of line character, '\n' has been encountered. Before

returning **getline()** places a '\0' to terminate the string. The contents of **str[]** are then written into the target file using the statement

```
outfile << str;
```

This process is repeated till all the lines from the source file have been read. Note that **getline()** function cannot use a hence we had to use the conventional **char** array to represent a line.

Record I/O

Suppose we wish to read/write employee records from/to file. Assume that each employee record contains information like name, age, basic salary and gross salary. If we use the overloaded operator << to write such records we would end up consuming more space for a record on the disk as compared to the space occupied by the same record in memory. This is because each number (integer or float) would be written as a character string rather than as binary bits of the number. For example 4250.55 would be written as a 7-byte string containing '4', '2', '5', '0', '.', '5', '5', instead of as a 4-byte float.

If a large number of employee records are to be stored in a file this would lead to a lot of wastage of precious disk space. If we are to avoid this we need to open the file in binary mode and then carry out the record input/output. The following program shows how this can be achieved.

```
#include  
#include  
#include  
using namespace std;
```

```
int main()
{
struct Employee
{
string name;
int age;
float basic, gross;
};
Employee e;

char ch = 'Y';

// create file for output
ofstream outfile;
outfile.open ("EMPLOYEE.DAT", ios::out | ios::binary);

while (ch == 'Y' || ch == 'y')
{
cout << endl << "Enter Name, Age, Basic Sal, Gross Sal:"
<< endl;
cin >> e.name >> e.age >> e.basic >> e.gross;
outfile.write (reinterpret_cast<
const char * > (&e),
sizeof (e));
cout << "Add another (Y/N) ";
cin >> ch;
```

```
}

outfile.close();

// open file for input
ifstream infile;
infile.open ("EMPLOYEE.DAT", ios::in | ios::binary);

cout << endl;
while (infile.read (reinterpret_cast< char * > (&e), sizeof (e)))
{
    cout << e.name << '\t' << e.age << '\t'
    << e.basic << '\t' << e.gross << endl;
}
return 0;
}
```

Here is some sample interaction with the program.

```
Enter Name, Age, Basic Salary, Gross Salary:  
Dinesh 23 3456.55 5644.45  
Add another (Y/N) Y
```

```
Enter Name, Age, Basic Salary, Gross Salary:  
Shirish 34 4455.55 5566.55
```

Add another (Y/N) Y

Enter Name, Age, Basic Salary, Gross Salary:

Leena 21 3455.55 3655.55

Add another (Y/N) N

Dinesh 23 3456.55 5644.45

Shirish 34 4455.55 5566.55

Leena 21 3455.55 3655.55

On execution, the program asks the user to enter employee records. Each record entered is written to file using the **ofstream::write()** function. Once all the records are written the file is closed. The same file is then opened for reading in binary mode and it is read record by record. Every record read is displayed on the screen.

We have used two new functions here— a member of and a member of These functions think about data in terms of bytes. They don't care how the data is formatted, they simply transfer a buffer full of bytes from and to a disk file. Consider the call to

```
outfile.write (reinterpret_cast<char * > (&e), sizeof (e));
```

Here we are trying to tell `write()` to write everything from the address given by `&e` up to the next `sizeof (e)` bytes. Note that it is necessary to cast the address passed to `write()` into a `const char` since `write()` doesn't know about an `employee`. To carry out this conversion from pointer of one type to an unrelated pointer type, C++ provides a `reinterpret_cast` operator.

The parameters passed to `read()` are similar to the ones passed to address of the data buffer and its length in bytes. This type pointer has to be casted into a `char`.

Random Access

In the previous section we wrote a program to write records into a file or read them back from it. At times we may wish to even modify an existing record. Once modification is over, we may wish to again add new records at the end of file, following which we may wish to read all records, or a particular record. This suggests that we should be able to move within a file to perform a desired operation. Let us now see how **iostream** library facilitates this.

Each stream has two long integers associated with it—a **get** pointer and a **put** pointer. The value present in the **get** pointer indicates the byte number in the file from where the next character would be read. Similarly, the value in the **put** pointer specifies where in the file the next write would take place. The term **pointer** in this context should not be confused with the normal C++ pointers. We can change the values of these pointers (using member functions) if we want to exercise control over where the next read/write should take place.

For example, while adding records to an existing file, we would like to carry out the addition at the end of existing records in the file. Likewise, while listing records we would like to start

reading records from the first record onwards. Similarly, while modifying records we would like to modify a record present at any place in the file. This means to carry out these operations we would be required to move the **get** and the **put** pointers.

The **istream::seekg()** and **ostream::seekp()** functions allow us to set the **get** and the **put** pointer respectively. To enquire their current positions we can use the functions **istream::tellg()** and

The seek functions come in two forms. The first form needs the absolute position within the file. For example,

```
file.seekg (1000L);
```

would position the **get** pointer at byte in the **file** stream.

The other form requires two arguments. The first argument specifies an offset from a location in the file, whereas the second specifies the location from which the offset is measured. The first argument can take positive or negative values. The second argument would be one of these— **cur** and **For example,**

```
file.seekg (-25L, ios::cur);
```

would position the **get** pointer **25** bytes before its **current** position. Likewise, **beg** and **end** refer to beginning and end of file respectively.

If we are to delete a record permanently from a file we have to open another file, write all records into it except the one that we want to delete, delete the original file and rename the new one back to original. Though apparently crude this is the way we can get rid of a record permanently. But this would be a pretty slow operation if the database already contains lot of records.

To improve the efficiency we may adopt a two-step policy —‘Delete’ and ‘Pack’. ‘Delete’ would only mark the record for deletion. The record would not be physically removed from the file. This marking can be done by placing a ‘*’ besides the record to be deleted. When we wish to ‘Pack’ the file, all starred records would be eliminated from the file permanently. Since on deleting the record it is only marked for deletion, we can provide a facility to even undelete the record. During recalling of a record the ‘*’ can be replaced by a space.

File Opening Modes

So far we have opened a file either for reading or for writing. There are several other modes in which a file can be opened. Each mode is defined by a bit in the `ios` class. We can combine these bits using the logical OR operator. [Table 8-4](#) shows the various possibilities.

Mode Bit	Result
in	Open a file for reading (default for <code>ifstream</code>)
out	Open a file for writing (default for <code>ofstream</code>)
ate	Start reading or writing at end of file (AT End)
app	Start writing at end of file
trunc	Truncate file to zero length if it exists
norcreate	Error when opening if file does not already exist
noreplace	Error when opening for output if file already exists, unless <code>ate</code> or <code>app</code> is set
binary	Open file in binary (not text) mode

Table 8-4. *File opening modes.*

If we want to preserve the file contents we should use `In` this case whatever we write to the file will be added at the end of the existing contents.

If we want to perform both input and output on the file in binary mode we can use the following open command.

```
fstream file;  
file.open (filename, ios::in | ios::out | ios::binary);
```

The vertical bars between the flags cause the bits representing these flags to be logically combined into a single integer, so that several flags can be applied simultaneously.

Quick now! Can you suggest the bit flag combination for opening a file for reading as well as writing? If the file does not exist a new one should get created, whereas if it is already existing then it should not get overwritten. The answer is

```
fstream file;  
file.open (filename, ios::in | ios::out | ios::noreplace);
```

String Streams

So far we have handled two types of iostreams—standard and file. The third type of iostream works with strings in memory and is called The beauty of iostreams is that even with this stream we can use the same reading and formatting functions. The class names for strstreams are similar to those for file streams. If we are to extract characters from a stringstream, we create an `istrstream`. If we are to insert characters into a stringstream, we create an `ostrstream`.

Readers familiar with Windows programming must have come across several functions for example) which needs a string as one of the parameters. If we are to display a char, a float an integer and a string through `MessageBox()` we are first required to convert them into strings, concatenate these strings and then pass the resultant string to `In`. In C++ we can use strstreams for this. This is shown in the program given below. We have formatted the data as it goes to the stringstream.

```
#include  
#include  
#include  
using namespace std;
```

```
int main()
{
char ch = 'Z';
int i = 350;
float a = 3.141528f;
char str[] = "strstreams at work";
ostrstream s;

s << "ch =" << ch << endl
<< "i =" << hex << i << endl
<< "a =" << fixed << a << endl
<< "str =" << str
<< ends;

cout << s.str() << endl;
return 0;
}
```

Here is the output of the program showing all elements neatly aligned.

```
ch = Z
i = 15e
a = 3.141528
```

`str = strstreams at work`

Here, first we have created an object `s` of type `Once`. Once this is done we can insert into it anything that we want, using the normal formatting ways that we used with

An important thing to remember about is that the zero terminator we normally need at the end of a character array is not inserted for us. We need to specifically insert it using the manipulator `Once`. Once the string is built, we can access it using

USING *ISTRSTREAM*

Instead of writing formatted data into string we can do the reverse too. We can extract bytes from string using `Here` is a simple program that shows how this can be achieved.

```
#include
```

```
#include
```

```
#include
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int age;
```

```

float salary;
string fname, mname, sname;
char str[] = "Sameer Shekhar Deshpande 35 12004.50";

istrstream s (str);
s >> fname >> mname >> sname >> age >> salary;
cout << fname << endl
<< mname << endl
<< sname << endl << age << endl << salary;

return o;
}

```

When we build the object **s** the constructor of **istrstream** gets called. It takes a pointer to a zero-terminated character array. Instead of an array we can also pass it a pointer to a zero-terminated string allocated on the heap.

Once the **istrstream** object is built, we can extract bytes from it until the '\0' present at the end of the string is encountered.

The statement

```
s >> fname >> mname >> sname >> age >> salary;
```

extracts different data items from the

Thus, strstreams offer a flexible and general approach to transforming character strings to typed values.

Object I/O

So far we have concentrated on I/O of standard types like **string** and their variants. We can also I/O objects of user-defined types. This can be accomplished by overloading the operators `>>` and `<<` for appropriate streams. For example, if we have an object **c1** of a user-defined class then we can read it and display it through the statements

```
Complex c1;  
cin >> c1;  
cout << c1;
```

The following program shows how this overloading can be achieved.

```
#include  
using namespace std;  
  
class Complex  
{  
private:  
    double real, imag;
```

```
public:  
Complex()  
{  
}  
Complex (double r, double i)  
{  
    real = r;  
    imag = i;  
}  
friend ostream& operator << (ostream& s, Complex& c);  
friend istream& operator >> (istream& s, Complex& c);  
  
};  
ostream& operator << (ostream& s, Complex& c)  
{  
    s << "(" << c.real << ", " << c.imag << ")";  
    return s;  
}  
istream& operator >> (istream& s, Complex& c)  
{  
    s >> c.real >> c.imag;  
    return s;  
}  
  
int main()  
{  
    Complex c1 (1.5, 2.5), c2 (3.5, 4.5), c3;  
    cout << "c1 =" << c1 << endl << "c2 =" << c2 << endl;
```

```
cout << "Enter a Complex number:";  
cin >> c3;  
cout << "c3 =" << c3 << endl;  
return o;  
}
```

You may note that the statements

```
cout << "c1 =" << c1 << endl << "c2 =" << c2 << endl;  
cout << "Enter a Complex number:";  
cin >> c3;  
cout << "c3 =" << c3 << endl;
```

are much more expressive and are similar to the way we would perform I/O with standard data types.

Here we have defined two **friend** functions **operator << ()** and **operator >>** We marked these functions as friends of the class **Complex** so that they can access **private** data of the **Complex** class. Since the **friend** declaration of these functions occurs only in **Complex** class and not in **istream** or **ostream** classes, these functions can access only the **private** data of **Complex** class.

The operator functions are not members of the **Complex** class. Hence the statement **cin >> c3** doesn't get converted into the form

```
cin.operator >> (c3);
```

The object on either side of **>>** gets passed to the operator function. Both are collected as references. This prevents creation of copies of these objects. The function returns the **istream** object by reference to permit cascading, as in

```
cin >> c4 >> c5;
```

Exactly same argument applies to the **operator << ()** function.

Serialization

We can also overload the `>>` and `<<` operators of **Complex** class discussed in the previous section to read / write objects from a file. However, if we do so, only the values of data members of the object will be written to the file. We will not be writing the object's type information, i.e. we will not be storing the fact that the two values being written for an object are and that these two belong to a **Complex** object.

If a program that reads the data from the file knows the object type to which the data corresponds, then it can read the data into an object of **Complex** type. But if it doesn't, then it would not be able to reconstruct the object.

Then there is a possibility that we store objects of different types in the same file. In such a case, we will not be able to distinguish them when we read them.

A solution to both these situations is object serialization and deserialization. During serialization along with object's data values, their types and the type of the object is also written to the file. During deserialization this type information is read

from the file and the object is reconstructed in memory using it.

C++ does not provide a built-in serialization / deserialization mechanism. To carry it out, we need to use a popular open source library called Boost. It provides support for serializing objects in text, binary and XML formats.

Error Handling during I/O

The I/O programs that we wrote so far were not sand-papered with error checks. Error checking, if any, was restricted only to reaching end-of-file condition. However, no professional program can stand the test of time unless exhaustive error checking is done in it. This involves error checking which opening a file and while reading/writing a file. The following program shows how these checks can be performed.

```
#include  
#include  
#include  
using namespace std;  
  
int main()  
{  
void report (ofstream&);  
  
ofstream file;  
file.open ("SAMPLE.TXT", ios::noreplace);  
  
if (! file)
```

```
{  
report (file);  
exit (1);  
}  
else  
{  
file << "Had cars been built like the OS," << endl  
<< "we would have had more hospitals than homes";  
if (! file)  
{  
report (file);  
  
exit (2);  
}  
}  
}  
file.close();  
return o;  
}  
void report (ofstream &file)  
{  
cout << endl << "Unable to open SAMPLE.TXT";  
cout << endl << "Error state =" << file.rdstate();  
cout << endl << "good() =" << file.good();  
cout << endl << "eof() =" << file.eof(); cout << endl << "fail()  
=" << file.fail(); cout << endl << "bad() =" << file.bad();  
}
```

So far when we opened a file for reading we assumed that it is existing. Similarly, when we opened it for writing we assumed that it could get created. It is not safe to rely on such assumptions if we want to make the program foolproof. Hence whenever we open a file we must check whether the file opening was successful or not. Also, when we write to the disk the possibilities like the disk getting full, or the disk being write-protected cannot be ruled out. While reading from a file if you don't consider the possibility of a hardware error, you are away from reality.

In the above program we have opened the file for writing. If this fails then we report an error. You may recall that in the statement

```
file.open ("SAMPLE.TXT", ios::noreplace);
```

file becomes zero (false) if we are unable to open the file. So we can check the status of **file** for success/failure in file opening:

```
if (! file)
```

If this condition is satisfied we have called the **report()** function where details of various error flags are printed. The

`ios` class maintains a state variable whose bits signify the various error states. The position of these bits and their meaning is given in [Figure](#)

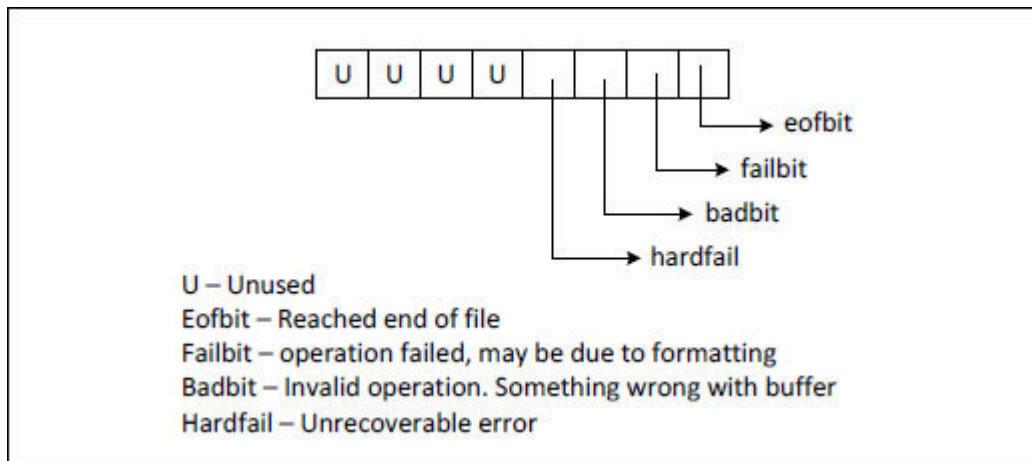


Figure 8-5. Meaning of bits in state variable.

The value of the state variable can be obtained through the function `eof()` and other functions like `fail()` and `bad()` return the status of other bits. The meanings of these functions are as under.

Function	Purpose
<code>eof()</code>	Returns true if EOF flag is set
<code>fail()</code>	Returns true if failbit or badbit or hardfail flag is set
<code>bad()</code>	Returns true if badbit or hardfail flag is set
<code>good()</code>	Returns true if everything OK; no flags set
<code>clear (int = 0)</code>	With no argument, clears all error bits; otherwise sets specified flags, as in <code>clear (ios::failbit)</code>

Table 8-5. Error checking functions.

Here's the output of the program when SAMPLE.TXT was existing and **noreplace** didn't permit replacement of it with a new file...

Unable to open SAMPLE.TXT"

Error state = 4

good() = 0

eof() = 0

fail() = 4

bad() = 4

The error state returned by **rdstate()** is 4. This indicates that the file couldn't be opened. The **good()** function returns 1 (true) only when no bits are set, so in our case it returned 0. We're not at EOF, so **eof()** returned 0. The **fail()** and **bad()** functions returned nonzero, since an error occurred. In a serious program some or all of these functions should be used after every I/O operation to ensure that things went as expected.

Interaction with File System

A C++ program may wish to access the file system of the machine on which it is running to perform various file and directory operations. These include operations like creation, renaming, deletion, navigation, etc. Different OS use different file systems. For example Windows uses NTFS, whereas Linux uses EXT3. As each file system manages files and folders differently, C++ programs that access file system resources are specific to that particular OS and file system.

Given below is a C++ program that shows how to perform common disk, file and directory operations for Windows OS using NTFS.

```
#include  
#include  
using namespace std;  
  
int main()  
{  
char path[MAX_PATH] = "";
```

```
unsigned long int spc, bps, nfc, tnc;
GetDiskFreeSpaceA (“C:\\”, &spc, &bps, &nfc, &tnc);

cout << “Sectors per Cluster =” << spc << endl;
cout << “Bytes per Sector =” << bps << endl;

cout << “Number of Free Clusters =” << nfc << endl;
cout << “Total Number of Clusters = << tnc << endl;

float capacity, avlbl, multiplier;
multiplier = spc * bps / 1024.of / 1024.of / 1024.of;
capacity = tnc * multiplier;
avlbl = nfc * multiplier;
cout << “Disk capacity =” << capacity << “GB” << endl;

cout << “Available free space =” << avlbl << “GB” << endl;

GetModuleFileNameA (NULL, path, MAX_PATH);
cout << “Exe file =” << path << endl;

GetCurrentDirectoryA (MAX_PATH, path);
cout << “Working Dir =” << path << endl;

cout << “Creating directory myDir...” << endl;
CreateDirectoryA (“myDir”, NULL);
```

```
cout << "Changing to directory myDir..." << endl;
SetCurrentDirectoryA ("myDir");
GetCurrentDirectoryA (MAX_PATH, path);
cout << "Working Dir =" << path << endl;

cout << "Changing to parent of directory myDir..." << endl;
SetCurrentDirectoryA ("..");
GetCurrentDirectoryA (MAX_PATH, path);
cout << "Current directory =" << path << endl;

cout << "Removing directory myDir..." << endl;
RemoveDirectoryA ("myDir");

cout << "Creating file tempFile..." << endl;
HANDLE hFile;
hFile = CreateFileA ("tempFile", GENERIC_WRITE,
FILE_SHARE_WRITE, NULL, CREATE_ALWAYS, 0, NULL);

FILETIME ftCreate, ftAccess, ftWrite;
SYSTEMTIME stUTC, stLocal;

// Retrieve file times for the file
GetFileTime (hFile, &ftCreate, &ftAccess, &ftWrite);

// Convert creation time to local time FileTimeToSystemTime
(&ftCreate, &stUTC);
```

```
SystemTimeToTzSpecificLocalTime (NULL, &stUTC, &stLocal);

cout << "File creation date = ";
cout << stLocal.wDay << "/" << stLocal.wMonth << "/"
<< stLocal.wYear << endl;
cout << "File creation time =";
cout << stLocal.wHour << ":" << stLocal.wMinute << endl;

return 0;
}
```

Here is the output of the program...

```
Sectors per Cluster = 8
Bytes per Sector = 512
Number of Free Clusters = 6715617
Total Number of Clusters = 40650239
Disk capacity = 155.068 GB
Available free space = 25.618 GB
Exe file = C:\Users\Kanetkar\Desktop\Sample\Debug\Sample.exe
Working Dir = C:\Users\Kanetkar\Desktop\Sample\Sample
Creating directory myDir...
Changing to directory myDir...
Working Dir = C:\Users\Kanetkar\Desktop\Sample\Sample\myDir
Changing to parent of directory myDir...
Current directory = C:\Users\Kanetkar\Desktop\Sample\Sample
```

```
Removing directory myDir...
Creating file tempFile...
File creation date = 25/3/2019
File creation time = 17:13
```

Notice the 'A' at the end of most functions used in this program. It stands for ANSI strings. It means these functions use the normal character strings. If we are to use Unicode character strings then we should prepend a 'L' before the string and replace 'A' with 'W' as shown in the following function call:

```
CreateDirectoryW (L“myDir”, NULL);
```

Exercise

[A] State whether the following statements are True or False:

In the **iostream** library the **ios** class is at the root.

Only for those manipulators which need an argument we need to include the file ‘iomanip.h’.

When we are using a manipulator we are in fact calling a member function.

It is possible to create your own manipulators.

cin and **cprn** are predefined stream objects.

Objects can read and write themselves.

strstreams are used for input / output from / to a string.

The **istream::getline()** function cannot tackle multi-words strings.

[B] Answer the following:

Draw a chart showing hierarchy of various classes in the **iostream** library.

How can the following statement work for testing the end of file, if **infile** is an **ifstream** object:

```
while (infile);
```

What do the **nocreate** and **noreplace** flag ensure when they are used for opening a file?

What problem do you think we would face if the following code is executed twice:

```
file.seekg (oL, ios::beg);
while (file.read (char *) &p, sizeof (p))
cout << t.name << endl << p.age;
```

How would you solve the problem ?

How would you obtain the value of a **state** variable?

[C] What will be the output of the following programs:

(a) #include

```
using namespace std;
```

```
int main()
```

```
{
```

```
char str[] = "The boring stuff";
```

```
char *p = "That's interesting";
```

```
cout << endl << str;
```

```
cout << endl << p;
```

```
cout << endl << (void *) str;
```

```
cout << endl << (void *) p;
```

```
return 0;
```

```
}
```

(b) #include

```
using namespace std;
```

```
int main()
```

```
{
```

```
int i = 650;
```

```
float a = 425.123;
```

```
cout << setiosflags (ios::showbase | ios::uppercase);
```

```
cout << hex << i << endl;
```

```
cout.precision (4);
cout << setiosflags (ios::showpoint) << a;
return o;
}
```

(c) #include
using namespace std;

```
int main()
{
    char str[] = "Just listing";
    cout.width (40);
    cout << str << endl;
    cout.setf (ios::left, ios::adjustfield);
    cout.width (40);
    cout << str;
    return o;
}
```

[D] Attempt the Following:

What is the limitation of the following overload operator function for displaying Complex objects?

```
ostream& operator << (ostream& s, Complex& c)
```

```
{  
s << ("(<< c.real <<", "<< c.imag <<")";  
}
```

Write a program that will create a directory path d₁/d₂/d₃. In d₃ create a file 'sample'txt'. Write your name and address in that file.

There are 100 records present in a file with each record containing a 6-character item code, a 20-character item name and an integer quantity. Write a program to read these records, arrange them in the ascending order and write them to the same file overwriting the earlier records.

KanNotes

Expectation from an IO System:

- I should be able to communicate with sources & destinations
- I should be able to I/O varied entities
- I should be able to communicate in multiple ways
- I should be able to deal with underlying file system

C++ solution - Perform all IO using Streams

Stream is a sequence of bytes that travel from source to destination over a communication path

Streams are implemented by classes in iostream library

Linking of Streams to physical devices is done by C++ IO system

C++ program performs IO by reading / writing from / to a stream

Benefits of using Streams

- Streams hide details of communication from programmer
- Methods are same, implementation changes as per device

Ready-made stream objects:

- cin, cout, cerr, clog - I/O of ASCII characters
- wcin, wcout, wcerr, wclog - I/O of Unicode characters
- cin - object of istream class
- cout, cerr, clog - objects of ostream class
- wcin - object of wistream class
- wcout, wcerr, wclog - object of wostream class

There are many stream manipulators that help you manipulate the output as it is sent to an output device

For manipulators that use arguments we should include the file ‘iomanip’

It is possible to create user-defined manipulators

Using classes present in ‘fstream’ it is possible to perform 3 types of file I/O operations:

- Character I/O - read/write file character by character
- Line I/O - read/write file line by line
- Record I/O - read/write file record by record

For random-access in a file following functions are useful:

- seekg(), seekp() - set get and put pointer in file
- tellg(), tellp() - return current location of get and put pointer in a file

There are multiple modes in which a file can be opened

To perform string I/O following strstream classes are useful:

- istrstream - to read data from a string
- ostrstream - to write data to a string

Object I/O can be performed by overloading the << and >> operators

Serialization - Writing an object to a file

Deserialization - Reading an object from a file

If file opening fails, then reason for failure is available in bits of a state variable

Interaction with file-system is platform dependant

We can create, change, navigate, delete files and directories through a C++ program

Advanced Features of C++

Though not very commonly required, you would be a more complete C++ programmer if you understand the advanced topics explained in this chapter.

Containership

friend Functions and *friend* Classes

One More Use of *friend* Function

A Word of Caution

The *explicit* Keyword

The *mutable* Keyword

Namespaces

Using a Namespace

Using Scope Resolution Operator

The *using* Keyword

RTTI

Typecasting in C++

static_cast

dynamic_cast

const_cast

reinterpret_cast

A Word of Caution

Pointers to Members

Exercise

KanNotes

This chapter covers a collection of topics that a beginner would usually not be faced with. But as you spend more time in C++ programming you would start appreciating the utility of these concepts. I plan to cover here topics like containership, friend functions, **explicit** and **mutable** keywords, namespaces, RTTI and the new casting syntax. Let's begin with containership.

Containership

We already know the inheritance relationship between classes. In inheritance if a class Y is derived from class X, we say that the new class Y is like that old class X. Or in other words 'Y is like X'. This is because the class Y has all the characteristics of X, and in addition some of its own. It is like saying a car is like an automobile—a car has the characteristics shared by all automobiles (has wheels and engine, runs on fuel etc.) but has some distinctive characteristics of its own (such as a four stroke engine and a steering wheel). For this reason, inheritance is often called a 'like a' relationship.

There is another type of relationship that two related classes may enjoy. This is known as 'has a' relationship. For example, suppose there are two related classes— **Carburettor** and a **Car**. These are related classes, since both are related to a vehicle. However, neither is **Carburettor** like a **Car** nor is **Car** like a **Carburettor**. In fact, the relationship between them can be— **Car** 'has a' or in other words, **Carburettor** is contained within a **Car**. Such a relationship is known as **Containership**.

Apart from inheritance, containership is another reuse mechanism offered by C++. Both these reuse mechanisms offer an object code level reuse, i.e. source code a class is not necessary for reusing it while creating another class. The program that illustrates the containership relationship between **Car** class and **Carburettor** class is given below.

```
#include
#include
using namespace std;

class Carburettor
{
private:
char type;
float cost;
string mfr;
public:
void setData (char t, float c, string m)
{
type = t;
cost = c;
mfr = m;
}
void displayData()
{
```

```
cout << type << endl << cost << endl << mfr << endl;
}
};

class Car
{
private:
string model;
string drivetype;
Carburettor cc; // embedded object
public:
void setData (char t, float c, string mf, string m, string d)
{
model = m;
drivetype = d;

cc.setData(t, c, mf);
}
void displayData()
{
cout << model << endl << drivetype << endl;
cc.displayData();
}
};

int main()
{
Car mycar;
mycar.setData ('A', 18500.00, "Mico", "Sports", "4-wheel");
}
```

```
mycar.displayData();  
return o;  
}
```

Given below is the output of the program.

Sports
4-wheel
A
18500
Mico

friend Functions and friend Classes

So far we have learnt that that only member functions can access **private** data of a class. This is the essence of data encapsulation. However, sometimes we have to make an exception to this rule to avoid programming inconvenience. At such times, we have to allow functions outside a class to access and manipulate the class's **private** data members. To achieve this, C++ provides a keyword called The following program will help you in understanding how this concept is put to work.

```
#include  
using namespace std;  
  
class Sample2;  
class Sample1  
{  
private:  
int data1;  
public:  
Sample1()  
{  
data1 = 100;
```

```
}

friend void accessOne (Sample1);
friend void accessBoth (Sample1, Sample2);
};

class Sample2
{
private:
int data2;
public:

Sample2()
{
data2 = 200;
}
friend void accessBoth (Sample1, Sample2);
};

void accessOne (Sample1 x)
{
cout << x.data1 << endl;
}

void accessBoth (Sample1 x, Sample2 y)
{
cout << x.data1 + y.data2 << endl;
}

int main()
{
Sample1 a;
Sample2 b;
```

```
accessOne (a);
accessBoth (a, b);
return o;
}
```

Here we have declared two classes **Sample1** and **Sample2**. The constructors in these classes initialize their **private** data items to a fixed value in **Sample1** and **200** in **Sample2**. We want the function **accessOne()** to have access to **private** data and function **accessBoth()** to have access to both class's **private** data. Hence, we have marked both as **friend** functions through the declarations

```
friend void accessOne (Sample1);
friend void accessBoth (Sample1, Sample2);
```

Of these the first declaration has been placed only in **Sample1** class since **accessOne** intends to access **private** data of **Sample1** alone. As against this, the second declaration has been placed in both classes as this function intends to access both class's **private** data. Note that these declarations can be placed either in the **private** section or the **public** section of the class.

A **Sample1** object is passed to **accessOne()** whereas **Sample1** and **Sample2** objects are passed to **accessBoth()**. Being **friend** functions **accessOne()** can

access **private** data, whereas **Sample2** can access **private** data of **Sample1** as well as Though the functions don't do much, I think the program serves to illustrate the concept.

Note that friend functions are global functions and are not members of any class. Hence a this pointer is never passed to them. Observe the declaration at the beginning of the program:

```
class Sample2;
```

This declaration is necessary since a class can't be referred to until it has been declared. Class **Sample2** is being referred to in the declaration of the function **accessBoth()** in class Hence **Sample2** must be declared before This declaration tells the compiler that the class **Sample2** is defined later.

This aspect of **friend** function—accessing **private** data of a class in a global function—has been used at two places in [Chapter](#) One is while defining the user-defined manipulator with arguments and another is while overloading the << and >> operators. You can revisit those sections now in light of things that we have learnt here.

Instead of marking a function as a friend function, we can mark a class as a **friend** class as shown below.

```
class Sample2;
class Sample1
{
private:
// some data members
public:
// some member functions
friend Sample2;
};
```

Here, since **Sample2** has been marked as a **friend** class, all its member functions can access private data of class

One More Use of friend Function

Friend functions are sometimes used to improve the versatility of overloaded operators. Let us try to understand this aspect of **friend** functions through the program given below.

```
#include
using namespace std;

class Example
{
private:
int i;
float j;
public:
Example (int ii = 0, float jj = 0.0f)
{
i = ii;
j = jj;
}
void showData()
{
cout << i << " " << j << endl;
}
```

```
Example operator * (Example e)
{
    Example temp;
    temp.i = i * e.i;
    temp.j = j * e.j;
    return (temp);
}
```

```
int main()
{
    Example e1 (10, 3.14f), e2 (2, 1.5f), e3, e4, e5;
    e3 = e1 * e2; // Example * Example is ok
    e4 = e1 * 2; // Example * int is ok
    e3.showData();
    e4.showData();
    // e5 = 2 * e1; // int * Example is not ok
    return 0;
}
```

In this program we have defined a class **Example** having an **int** and a **float** as its **private** data members. It has two member function—a constructor and an overloaded ***** operator to multiply two objects of type

The statement

```
e3 = e1 * e2;
```

internally becomes

```
e3 = e1.operator * (e2);
```

This calls the overloaded operator functions and returns the product of the two **Example** objects.

Similarly, the statement

```
e4 = e1 * 2;
```

internally becomes

```
e4 = e1.operator * (2);
```

The **2** being passed to the operator function needs to be converted to an object of the type. The compiler will use the constructor to convert this **int** to an and then carry out the multiplication.

Would the following statement work?

```
e5 = 2 * e1;
```

No, because internally this would become

```
e5 = 2.operator (e1);
```

which would not make any sense since **2** is not an object to begin with. The compiler can't handle this situation, hence flashes an error. A **friend** function can help us out in this situation. The following program shows how this can be achieved.

```
#include
using namespace std;

class Example
{
private:
int i;
float j;
public:
Example (int ii = 0, float jj = 0.0f)
{
i = ii;
j = jj;
```

```
}

void showData()
{
    cout << i << "\t" << j << endl;
}

friend Example operator * (Example, Example);
};
```

```
Example operator * (Example k, Example l)
{
    Example temp;
    temp.i = k.i * l.i;
    temp.j = k.j * l.j;
    return (temp);
}
```

```
int main()
{
    Example e1 (10, 3.14f), e2 (1, 1.5f), e3, e4, e5;
    e3 = e1 * 2;
    e4 = 2 * e2;
    e5 = e1 * e2 * 2;
    e3.showData();
    e4.showData();
    e5.showData();
    return 0;
}
```

Note that, in the previous program the **operator *()** function took only one argument, whereas in this program it takes two. This is because the operator function is no longer a member function of the class. It is a **friend** of the class. Thus the statement

```
e3 = e1 * 2;
```

doesn't take the form **e3.operator * (2)**.

This program shows that using **friend** permits the overloaded operators to be more versatile.

A Word of Caution

Usage of **friend** functions is a bit controversial. Though it adds flexibility to the language and makes programming convenient in certain situations, it goes against the philosophy that only member functions can access a class's **private** data.

However, C++ is not a pure OO language. Facility to access **private** data from outside the class through is one of the reasons for it. But then C++ was designed to solve practical programming problems, not to win prizes for purity. If violate a few lofty ideals but serves programming convenience, so be it.

On the flip side, if you find yourself using many then maybe there is a basic flaw in the design of your program and you would be better off redesigning it.

The explicit Keyword

In [Chapter 5](#) we saw that data conversion from standard type to user-defined type is possible through conversion operator and the class's constructor. However, there may be some conversion which you want should **not** take place. It's easy to prevent conversion performed by the conversion operator—just don't define the operator.

However, the same doesn't apply to the constructor. We may want it for building the object. At the same time you may want that it should not get used for carrying out conversions. Through the **explicit** keyword we can prevent such unwanted conversions. The following program shows the use of this keyword.

```
#include  
using namespace std;  
  
class Complex  
{  
private:  
float real, imag;
```

```
public:  
Complex (float r = 0.0f, float i = 0.0f)  
{  
real = r;  
imag = i;  
}  
Complex operator + (Complex c)  
{  
Complex t;  
t.real = real + c.real;  
  
t.imag = imag + c.imag;  
return t;  
}  
void display()  
{  
cout << real << " " << imag << endl;  
}  
};  
  
int main()  
{  
Complex c1 (1.5, 3.5), c2;  
c2 = c1 + 1.25f;  
c2.display();  
return 0;  
}
```

In the statement

```
c2 = c1 + 1.25f;
```

overloaded **operator + ()** function should get called. The compiler finds that this function expects a **Complex** argument, whereas we are trying to pass a **float** to it. Hence, it looks for a conversion function which can convert a **float** to The constructor meets this requirement. Hence the compiler decides to call it. This is good. But had we written the statement as

```
c2 = 1.25f + c1;
```

it would not work.

This is unacceptable, since addition all our life has been a commutative operation. So for consistency's sake, we should ensure that both the following statements should not work.

```
c2 = c1 + 1.25f;
```

```
c2 = 1.25f + c1;
```

We can ensure this by marking the constructor as **explicit** as shown below.

```
explicit Complex (float r = 0.0f, float i = 0.0f)
{
    real = r;
    imag = i;
}
```

Now any attempt to carry out implicit conversions from float to Complex the compiler would report that it cannot do the conversion.

Note that the **explicit** keyword works only with the constructors.

The **mutable** Keyword

When we create a **const** object none of its data members can change. In a rare situation, however, you may want some data member should be allowed to change despite the object being **const**. This can be achieved by using the **mutable** keyword as shown in the following program.

```
#include  
#include  
using namespace std;  
  
class Car  
{  
private:  
    string model;  
    mutable string owner;  
    int yrOfMfg;  
    string regNo;  
public:  
    Car (string m, string o, int y, string r)  
    {  
        model = m;  
        owner = o;
```

```
yrOfMfg = y;
regNo = r;
}
void changeOwner (string o) const
{
owner = o;
}
void changeModel (string m)

{
model = m;
}
void display() const
{
cout << model << endl << owner << endl
<< yrOfMfg << endl << regNo << endl << endl;
}
};

int main()
{
const Car c1 ("VX", "Fundu", 2000, "MH31-G6175");
c1.display();
c1.changeOwner ("MahaFundu");
c1.display();
// c1.changemodel ("AX"); // reports error
c1.display();
return 0;
```

}

When a car is sold its owner will change, whereas, rest of its attributes should remain same. Since the object **c1** is declared as none of its data members can change. An exception is however made in case of **owner** since its declaration is preceded by the keyword **mutable**. The change is brought about through the function **changeOwner**. Try removing the comment in **changeOwner**. This would result in an error as the **model** data member has not been declared as **mutable** and hence cannot be changed. Had **c1** been a non- **const** object we would have been allowed to change the **owner** as well as the **model**.

Namespaces

Trivial it may seem, but creating names is one of the most basic activities in programming. Variable names, array names, function names, structure names, class names, union names, enumeration names, all fall under one general category—While writing big programs involving several programmers, things are likely to go out of hand if proper control is not exercised over visibility of these names. For example, consider the following two header files.

```
// mylib.h
char fun1();
void display();
class BigNumber {...};

// somelib.h
class BigNumber {...};
void display();
```

If both these header files are included in a program, there would be a clash between the two **BigNmuber** classes and the two **display()** functions. This would lead to a conflict, hence would be reported as errors.

One solution to this could be to create long names such that there is a less likelihood of a clash. But then you are required to type these long names. Moreover, it is a compromise solution invented by programmers and not a language supported solution. C++ offers a better solution to such problems through a keyword called **namespace**.

C++ provides a single global namespace. We can subdivide the global namespace into more manageable pieces using the **namespace** feature of C++. The following code shows how.

```
// mylib.h
namespace myLib
{
    char fun1() {...};
    void display() {...};
    class BigNumber {...};
}
```

```
// somelib.h
namespace someLib
{
    class BigNumber {...};
    void display() {...};
}
```

Now the class names will not clash because they become **myLib::BigNumber** and respectively. Same thing would happen to the function names. They would become **myLib::display()** and thereby avoiding a clash.

Thus, it is now possible to use the same name in separate namespaces without conflict. As long as they appear in separate namespaces, each name will be unique because of the addition of the **namespace** identifier.

Now a few points worth noting about the namespaces.

The syntax for creation of a **namespace** is similar to that of a class except that in a namespace there is no semicolon beyond the closing brace.

Declarations that fall outside all namespaces are still members of the global namespace.

A namespace definition can be continued over multiple header files as shown below:

```
// mylib.h
namespace myLib
```

```
{  
char fun1() {...};  
void display() {...};  
}
```

```
// mylib1.h  
namespace myLib  
{  
extern int var1;  
void display() {...};  
}
```

When a namespace is continued in this manner, after its initial definition, the continuation is called an **extension-namespace**.

We can give an alternative name for a namespace. It is known as namespace-alias. It prevents typing of unwieldy names as shown below.

```
namespace hardAndSoftLibrary  
{  
class hwlItem {...};  
class swlItem {...};  
}  
namespace hws = hardAndSoftLibrary;
```

A global namespace-name cannot be the same as any other global entity name in a given program.

Members of a named namespace can be defined outside the namespace in which they are declared. In this case it is necessary to explicitly qualify the name being defined as shown below:

```
namespace mine
{
void fun1();
}
void mine::fun1()
{
```

If a name in a namespace is being defined outside it, then the definition must appear after the point of declaration.

```
void mine::fun2() // error, fun2() is not yet a member of mine
{
```

```
namespace mine
{
```

```
void fun2();  
}
```

A namespace definition can only appear at the global scope.
Thus the following code would cause an error.

```
int main()  
{  
namespace local // error: not at global scope  
{  
}  
return 0;  
  
}
```

A namespace definition can be nested within another namespace definition. For example:

```
namespace outer  
{  
int n = 6;  
int fun2();  
namespace inner  
{  
float a = 3.14;  
}  
}
```


Using a Namespace

There are two ways to refer to a name within a namespace.
These are

Using the scope resolution operator

Through the **using** keyword

Let us understand these methods one by one.

USING SCOPE RESOLUTION OPERATOR

We can specify any name in a namespace using the scope resolution operator, as shown below.

```
#include  
#include  
using namespace std;  
  
namespace mine  
{
```

```
class MyClass
{
private:
int yr;
public:
void changeYear();
};

class YourClass;
void fun1();
}

void mine::MyClass::changeYear()
{
yr = 2000;
cout << "Years don't change";

}

class mine::YourClass
{
public:
YourClass();
void show();
};

mine::YourClass::YourClass()
{
cout << "Reached YourClass's o-arg ctor" << endl;
}

void mine::YourClass::show()
{
```

```
cout << "Do it. Then don't think about it" << endl;
}

void mine::fun1()
{
    cout << "Be impulsive. Exercise Caution" << endl;
}

int main()
{
mine::MyClass m;
m.changeYear();
mine::fun1();
mine::YourClass y;
y.show();
return 0;
}
```

To reach a name in a namespace using the scope resolution operator is a bit tedious. Better solution comes in the form of the **using** keyword.

THE **USING** KEYWORD

Instead of being required to use the scope resolution operator before every name, the **using** keyword allows you to import an

entire namespace at once. Let us write the previous program on these lines and appreciate the convenience of this method.

```
#include
#include
using namespace std;

namespace mine
{
    class MyClass
    {
        private:
            int yr;
        public:
            void changeYear();
    };
    class YourClass;
    void fun1();
}
void mine::MyClass::changeYear()
{
    yr = 2000;
    cout << "Years don't change" << endl;
}
class mine::YourClass
{
```

```
public:  
YourClass();  
void show();  
};  
mine::YourClass::YourClass()  
{  
cout << "Reached YourClass's 0-arg ctor" << endl;  
}  
void mine::YourClass::show()  
{  
cout << "Do it. Then don't think about it" << endl;  
}  
void mine::fun1()  
{  
cout << "Be impulsive. Exercise Caution" << endl;  
}
```

```
int main()  
{  
using namespace mine;  
MyClass m;  
m.changeYear();  
fun1();  
YourClass y;  
y.show();  
return 0;  
}
```

The **using** keyword declares all the names in the namespace to be in the current scope. So we can use the names without any qualifiers.

RTTI

RTTI stands for **Run Time Type**. As the name suggests, it lets us find out the type of object at runtime. As we saw in [Chapter 7](#), while using virtual functions, it is necessary to identify the types at runtime to bind the function call to function of the correct object. Apart from this, in a complicated program we may wish to log the names of the types during runtime.

The type can be obtained at runtime using the **typeid** operator. The following program shows how to use it.

```
#include  
#include  
using namespace std;  
  
class Base  
{  
public:  
    virtual void fun1() {}  
};  
class MyClass: public Base {};  
class YourClass: public Base {};
```

```
int main()
{
Base *b1, *b2;
MyClass m;
YourClass y;

b1 = &m;
b2 = &y;
cout << typeid (b1).name() << endl;
cout << typeid (b2).name() << endl;
cout << typeid (*b1).name() << endl;

cout << typeid (*b2).name() << endl;

if (typeid (*b1) == typeid (*b2))
cout << "Equal" << endl;
else
cout << "Unequal" << endl;

cout << typeid (45).name() << endl;
cout << typeid ('4').name() << endl;
cout << typeid (4.5 + 2.2).name() << endl;

return 0;
}
```

Here is the output of the program...

```
class Base *
class MyClass
class YourClass
Unequal
int
char
double
```

In this program we have derived two classes— **MyClass** and **YourClass**— from a base class called Then we have created two base class pointers **b1** and **b2**, and two derived class objects **m** and We then did upcasting through the statements

```
b2 = &y;
b1 = &m;
```

Then we used **typeid** operator to obtain the types of pointers as well as types to which they are pointing to. **typeid** operator finds type of its operand, stores it in a **const** object of the type **typeinfo** and returns its reference. Using this reference we can call the function **name()** to obtain the type name as a string.

The types returned by **typeid** can also be compared using `==` and `!=`. **typeid** can also report the types of standard data types, or even expressions.

Typecasting in C++

C++ is a strongly-typed language. It means if there is a mismatch in type an error would be reported. But at times we want a value to be interpreted differently than its current type. At such times we need to explicitly carry out conversion from one type to another using typecasting operation.

Code snippet given below shows these conversions.

```
float x = 3.14;  
int y;  
y = (int) x; // C style casting  
y = int (x); // C++ style casting
```

This style of typecasting is alright for fundamental data types. However, if we use them indiscriminately on unrelated user-defined types, the code may turn out to be grammatically correct, but may cause runtime errors or produce unexpected results.

To ensure that such errors are reported at compile-time itself, C++ provides 4 typecasting operators. These type-conversion

operators and their syntax are given below.

```
static_cast < type > (expression)
dynamic_cast < type > (expression)
reinterpret_cast < type > (expression)
const_cast < type > (expression)
```

Let us now understand them one by one.

STATIC_CAST

A **static_cast** is used for conversions that are well-defined. These include:

Narrowing conversions

Conversions to **void ***

These are shown in the following program. Read the comments in the program carefully.

```
#include
using namespace std;
```

```
class Base {};
class Derived: public Base {};
class Sample {};

int main()
{
    int i = 10;
    long l;
    float f;
    char str[] = "Nagpur";

    // narrowing conversions
    i = static_cast < int > (l);
    i = static_cast < int > (f);

    // conversion to void pointer to print address
    void *vptr;
    vptr = static_cast < void * > (str);
    cout << vptr << endl;

    // conversion of unrelated types
    Sample *sptr;
    sptr = (Sample *) baseptr;
    // sptr = static_cast < Sample * > (baseptr); // error

    return 0;
}
```

Most of the program is self-explanatory. We would concentrate only on the last two statements of the program. When we try to cast a pointer to **Base** into an unrelated pointer to **Sample** using a traditional cast no error is reported. However, when we attempt to do the same using an error is flashed. This means that **static_cast** won't allow us to cast indiscriminately, making it safer than traditional casting.

DYNAMIC_CAST

This casting operator can only be used only with pointers and references to classes or with void*. It is called 'dynamic' because it checks at runtime whether we can 'safely' assign a pointer type to a pointer of another type. If not, it sets up a null value in the pointer, which we can check before using the pointer to call functions.

Let us understand this dynamic checking aspect using a program.

```
#include  
#include  
using namespace std;
```

```
class Employee
{
private:
string name;
public:
Employee (string n)
{
name = n;
}
virtual void showData()
{

cout << "Name:" << name << endl;
}
};

class Manager: public Employee
{
private:
double commision;
public:
Manager (string n, double comm): Employee (n)
{
commision = comm;
}
virtual void showData()
{
cout << "Commision:" << commision << endl;
}
```

```
}

};

int main()
{
Employee e1 ("Dinesh");
Manager m1 ("Suresh", 3000.0);

Employee *pemp1, *pemp2;
Manager *pmgr;

pemp1 = &e1;
pemp2 = &m1;
pmgr = dynamic_cast < Manager * > (pemp1);
cout << "pmgr =" << pmgr << endl;
if (pmgr)
pmgr -> showData();

else
cout << "pmgr contains null on downcasting" << endl;

pmgr = dynamic_cast < Manager * > (pemp2);
cout << "pmgr =" << pmgr << endl;
if (pmgr)
pmgr -> showData();
else
cout << "pmgr contains null on downcasting" << endl;
```

}

The need for **dynamic_cast** generally arises when we wish to perform derived class operation on a derived class object, but we have only a pointer or reference to the base class. In such a case, we downcast the base class pointer to derived class and then perform the operation.

When we do this downcasting using it checks whether the object bound to the pointer is an object of the target type or not. If not, it sets a value 0 in the pointer.

Let us understand this statement with reference to our program. Consider the following statements:

```
pmgr = dynamic_cast < Manager * > (pemp1);
pmgr = dynamic_cast < Manager * > (pemp2);
```

Here **pemp1** contains **Employee** object's address, whereas **pemp2** contains **Manager** object's address. So when we attempt to cast **pemp1** into **pmgr** (the target pointer type), casting cannot be done since **pemp1** contain's **Employee** object's address. So, in this case **pmgr** is set to null.

The second cast operation works, as in this case **pmgr** is pointing to an object of type

Let us now see a few quick additional tips about dynamic casting.

If we used **static_cast** or regular casting in the program, then runtime checking would not have happened. A **static_cast** returns as if nothing were wrong; this can be misleading.

Although **dynamic_cast** conversions are safer, **dynamic_cast** only works on pointers or references, and the run-time type check is an overhead.

Since upcasting is an implicit conversion it doesn't matter whether we use **dynamic_cast** or no casting at all.

If references are used instead of pointers and the dynamic casting fails, then an exception of type **bad_cast** is thrown.

CONST_CAST

This casting operator permits us to manipulate constness of an object. Using this operator we can convert a **const** to a non- **const** or vice-versa.

```
#include
using namespace std;

void display (char *);
int main()
{
const char *ptr = "What's up?";
display (const_cast < char * > (ptr));
return 0;
}

void display (char *p)
{
cout << p << endl;
}
```

Here since **display()** is expecting a **char** we need to convert the **const char *** into a **char *** before passing it to

Note that removing the constness of an object is not a good idea. The above program worked because **display()** did not attempt to manipulate the object pointed to by If we attempt to do so through a statement like

```
*p = 'H';
```

the program would compile, but would cause an undefined behavior at runtime.

REINTERPRET_CAST

This operator can be used to carry out following two conversions:

Conversion from one pointer type to another pointer type

Conversion of a pointer into an integer and vice versa

The first conversion merely copies the value in one pointer to another, leaving it for the programmer to decide whether this conversion is meaningful or not. This casting mechanism is the least safe and more often than not a source of bugs.

A WORD OF CAUTION

Whenever you feel the need to use the explicit type conversion you should take time to reconsider it. You would find that in many situations it could be completely avoided. In others, it can be localized to a few functions within the program. Always remember that whenever you are using a cast you are breaking the type system. And that is fraught with dangers.

Pointers to Members

We know that to access a structure member we use a ‘.’ or a ‘->’ operator. Also, to dereference a pointer we use the * operator. Following code snippet shows this at work.

```
int i;
int *ptr;
struct Emp
{
    char name;
    int age;
};
Emp e, *eptr;
ptr = &i;
cout << *ptr; // dereferencing
cout << e.name; // access
cout << eptr -> name; // access
```

If we want we can set up pointers to particular members of a structure. Since the elements of a structure are laid out in contiguous memory locations, the address of any structure element is really an offset from the starting address of the structure. Now to access the structure element through this

pointer we need ‘.’ or ‘->’ to reach the element and ‘*’ to dereference the pointer. To carry out the access and the dereferencing simultaneously, C++ provides two new operators: ‘.*’ and ‘->*’. These are known as **pointer to member** operators. The following program shows their usage.

```
#include
using namespace std;

struct Sample

{

int a;
float b;
};

int main()
{
int Sample::*p1 = &Sample::a;
float Sample::*p2 = &Sample::b;

Sample so = {10, 3.14f};
cout << so.*p1 << endl << so.*p2 << endl;

Sample *sp;
sp = &so;
```

```

cout << sp->*p1 << endl << sp->*p2 << endl;

// we can even assign new values
so.*p1 = 20;
sp->*p2 = 6.28f;
cout << so.*p1 << endl << so.*p2 << endl;
cout << sp->*p1 << endl << sp->*p2 << endl;

return o;
}

```

Note the definition of the pointers **p1** and

```

int Sample::*p1 = &Sample::a;
float Sample::*p2 = &Sample::b

```

Consider the part before the assignment operator. The stars indicate that **p1** and **p2** are pointers. **Sample::** indicates they are pointers to an **int** and a **float** within We have also initialized these pointers while declaring them, with addresses of **a** and **b** respectively.

Really speaking there is no “address of” **Sample::a** because we are referring to a class and not to an object of that class. **&Sample::a** merely produces an offset into the class. The

actual address would be produced when we combine that offset with the starting address of a particular object.

Hence **&Sample::a** is nothing more than the syntax of pointer to member. If we use **p1** and **p2** with one object we would get one set of values, if we use it with another we would get another set of values. This is what is shown towards the end of the program, where we have built an array of objects and accessed all objects' elements using **p1** and Moral is that the pointers to members are not tied with any specific object.

To the left of '`*`' there should always be a structure variable (object) or a reference and to the left of '`->*`' there should always be a pointer to a structure.

That brings us to an important question—isn't it a bad idea to make data public as we did in the above program. Yes, it is. Hence, pointer to members are more commonly used with member functions (which are usually rather than the data members of a class).

By using pointers to members we can have the flexibility of choosing a member function to be called, at run time. This permits us to select or change the behavior at run time. Sounds abstract? Well, you would soon understand it. For that firstly we will have to understand a pointer to a function. The following program shows how it works.

```
#include
using namespace std;

int main()

{
void fun (int, float);
void (*p) (int, float);
p = fun;
(*p) (10, 3.14f);
return 0;
}
void fun (int a, float b)
{
cout << a << endl << b << endl;
}
```

Here **p** is a pointer to a function that receives an **int** and a **float** and returns a Note that the parentheses around ***p** are necessary. In the declaration would would become prototype declaration of a function **p** which receives an **int** and a **float** and return a **void**

We have initialised **p** to address of function Mentioning the function name without a pair of parentheses gives its address

in memory. If we want, we can still use the **&** operator to get the address as in

```
p = &fun;
```

The syntax for calling **fun()** using **p** is as follows:

```
(*p)(10, 3.14);
```

If we want we can have even an array of pointers to functions and then call each function in turn as shown in the following code snippet:

```
void (*p[3])(int, float) = {fun1, fun2, fun3};  
for (int i = 0; i <= 2; i++)  
  
(*p[i])(14 + i, 5.54 + i);
```

One basic condition for this code to work is that the prototypes of functions **fun2()** and **fun3()** must be same, otherwise we cannot gather their addresses in the array

Let's now go a step further. Let's make **fun2()** and **fun3()** **public** members of class **sample** and then try to call them in a manner similar to the one shown above.

```
#include
using namespace std;

class Sample
{
public:
void fun1()
{
cout << this << "In fun1" << endl;
}
void fun2()
{
cout << this << "In fun2" << endl;
}
void fun3()
{
cout << this << "In fun3" << endl;
}
};

int main()
{
Sample so[4];
void (Sample::*p[3])() = {
    &Sample::fun1,
```

```
&Sample::fun2,
&Sample::fun3
};
for (int j = 0; j <= 3; j++)
{
for (int i = 0; i <= 2; i++)
(so[j].*p[i])();
}
return o;
}
```

Using the ‘.*’ syntax now we can call different member functions using different objects. While doing so, we have also printed the address of each object with reference to whom the member function is being called. The output looks like this...

```
002BF994 In fun1
002BF994 In fun2
002BF994 In fun3
002BF995 In fun1
002BF995 In fun2
002BF995 In fun3
002BF996 In fun1
002BF996 In fun2
002BF996 In fun3
002BF997 In fun1
002BF997 In fun2
002BF997 In fun3
```


Exercise

[A] State whether the following statements are True or False:

The phenomenon of writing a class within a class is known as composition.

Composition and containership is one and the same thing.

Composition and Inheritance both promote reuse of code.

A function that is a **friend** of a class can access **private** data member of a class but cannot manipulate them.

If an entire class is made a **friend** of another then all the member functions of this class can access the **private** data member of the original class.

Unless you have the source code of a class you cannot declare a function to the friend of that class.

Using a smart pointer we can iterate through a container.

Using a smart pointer we can make an object appear like a pointer.

The overloaded operator -> is called a smart pointer operator.

The pointer to member operator can be used to access particular data members within a class.

Pointers to members are not tied with any specific object.

A namespace definition can be continued over multiple header files.

[B] Answer the following:

Is it necessary that while using pointer to member with data, the data must be made

When should the **explicit** keyword be used in the constructor?
Can it be used with any other function?

How would you change a data member of the **const** object?

If two header files contain same names how would you avoid a clash of names if both the files are to be included?

How would you give an alternate name to a namespace?

[C] Attempt the following:

Write a program which has a class called Derive two classes **derived1** and **derived2** from it. Store address of derived class objects into base class pointers. Using **typeid()** and **dynamic_cast** identify the type of the object at run-time.

Write a program that will convert an integer pointer into an integer and vice-versa.

KanNotes

Containership or Composition is an object code level reuse mechanism. It lets us create a sub-object within an object

A function can be declared as a friend of a class

A friend function can access private data of a class of which it is a friend

Even though a friend declaration is done within a class, its definition must be done outside the class

A friend declaration can be made in private or public section of a class

A this pointer is never passed to a friend function

A function can be made a friend of multiple classes

If a function is a friend of multiple classes then the friend declaration is necessary in all these classes

If a class is declared as a friend of another class then all the member functions of this class can access private members of the class which contains the friend declaration

An overloaded operator function may be declared as a friend function to improve its versatility

Friend functions break the type system of C++, so use it sparingly and carefully

If a constructor is marked as explicit it cannot be used by the compiler to make implicit conversions

If a data member of a const object is to be kept changeable, mark it as such using the keyword mutable

Namespaces act as a container for related types

```
namespace
```

```
{  
class A {}; class B {};  
}
```

A namespace can be split over multiple files since it is a logical container

Nested namespaces are legal

A type in a namespace can coexist with another type with the same name, but belonging to a different namespace

Namespaces can be aliased as in

```
namespace s = std;
```

There are two ways to access a name in a namespace

1) Using the :: operator 2)

Using the using keyword

It is possible to obtain the type of an entity at runtime using the typeid operator

typeid operator can be used with an object, pointer, variable, or an expression as in typeid (45), typeid (ptr), typeid (*ptr)

Type returned by typeid can be compared with another returned type using == and

```
!= if (typeid (45) == typeid (i))
```

Following type conversions happen implicitly:

- 1) Widening conversion like assigning an int to a double
- 2) Assigning derived class object's address to pointer to base class

Explicit conversion can be done using 4 typecasting operators

- 1) static_cast
- 2) dynamic_cast
- 3) const_cast
- 4) reinterpret_cast

static_cast conversion is used to do:

- 1) Narrowing conversion
- 2) Pointer conversion to void * to print address of an array

dynamic_cast conversion is used to do downcasting of pointers

While using dynamic_cast at runtime it can be checked whether the conversion is safe or not

const_cast conversion is used to convert a const to a non-const and vice versa

reinterpret_cast is used to

- 1) Convert a pointer to an integer and vice versa
- 2) Convert between unrelated pointer types

This conversion is unsafe

'.' and '>' are called member access operators

'.' and '>' are used to carry out the access and the dereferencing simultaneously

'.' and '>' can be used to access public member functions of a class

Templates

Instead of writing and maintaining multiple similar functions or classes which work on different data types, we can write generalized functions / classes and leave it for compiler to handle the specializations. This chapter shows how this can be done.

Function Templates

What Happens at Compile Time?

Function Templates for User-defined Types

One More Function Template

Function Template Override

Multiple Argument Types

Templates versus Macros

[A Template-based Sort](#)

[Class Templates](#)

[A Linked List Class Template](#)

[Tips about Templates](#)

[Variadic Templates](#)

[Applications of Templates](#)

[Exercise](#)

[KanNotes](#)

Templates are a mechanism that make it possible to use one function or class to handle many different data types. By using templates, we can design a single class/function that operates on data of many types, instead of having to create a separate class/function for each type. When used with functions they are known as **function templates**, whereas when used with classes they are called **class templates**. We would first look at function templates and then go on to class templates.

Function Templates

Suppose we wish to write a function that returns minimum of two numbers passed to it. This will be a very simple function. If the number parameters are of different types in different calls, then we will have to write multiple overloaded versions of it as shown below.

```
// min for ints
int myMin (int a, int b)
{
    return (a < b) ? a : b;
}

// min for longs
long myMin (float a, float b)
{
    return (a < b) ? a : b;
}

// min for chars
char myMin (char a, char b)
{
    return (a < b) ? a : b;
}

// etc...
```

This approach has three disadvantages. These are

We have to repetitive code in multiple functions.

Multiple function versions consume more disk space.

If we decide to make a change in one function, we have to remember to make similar change in all versions of that function. Thus maintenance of the program becomes tedious.

Won't it be nice if we could write such a function just once, and make it work for many different data types. This is exactly what function templates do for us.

The following program shows how to write the **myMin()** function as a template, so that it will work with any standard type. We have invoked this function from **main()** for different data types.

```
#include  
using namespace std;  
template < class T >  
T myMin (T a, T b)  
{
```

```
return (a < b) ? a : b;
}

int main()
{
    int i = 10, j = 20;
    cout << myMin (i, j) << endl;

    float a = 3.14f, b = -6.28f;
    cout << myMin (a, b) << endl;

    char ch = 'A', dh = 'Z';
    cout << myMin (ch, dh) << endl;

    double d = 1.1, e = 1.11;
    cout << myMin (d, e) << endl << endl;

    return 0;
}
```

Here's the output of the program...

```
10
-6.28
A
```

```
1.1
```

As you can see, the **myMin()** function now works with different data types that we use as arguments. Isn't this code reuse? Yes, but of a different type. Inheritance and composition provide a way to reuse **object** code. Templates provide a way to reuse the **source** code. Templates can significantly reduce source code size and increase code flexibility without reducing type safety.

Let us now understand what grants the templated function the flexibility to work with different data types. Here is the definition of the **myMin()** function:

```
template < class T >
T mymin (T a, T b)
{
    return (a < b) ? a : b;
}
```

This entire syntax is called a function template. In a function template a data type can be represented by a name in our case) that can stand for any type. There's nothing special about the name We can use any other name like etc. **T** is known as a template argument. Throughout the definition of the function, wherever a specific data type like **int** would ordinarily be written, we substitute the template argument,

WHAT HAPPENS AT COMPILE TIME?

Just seeing the function template doesn't swing the compiler into any real action, except for memorizing it for future use. The compiler cannot generate any code as yet because it doesn't know as yet what data type the function will be working with. The code generation takes place when the function is actually called from within the program through statements like:

```
cout << endl << myMin (i, j);
```

When the compiler sees such a function call, it knows that the type to use is because that's the type of the arguments **i** and **j**. Now it generates a specific version of the **myMin()** for type replacing every **T** with an **i**. This process is often known as **instantiating** the function template. The compiler also replaces the call to **myMin (i, j)** with a call to the newly instantiated function.

Similarly, the expression **myMin (a, b)** causes the compiler to generate a version of **myMin()** that operates on type **float** and a call to this function. Likewise, **myMin (d, e)** call generates a function that works on type **Note**. That the compiler generates only one version of **myMin()** for each data type irrespective of the number of calls that have been made for that type.

Do templates help us save memory? Not really—because, even when we use templates the four functions (for **float** **char** and **do**) get generated. The advantage is we are not required to create or maintain them. The compiler creates them from our generic version. This makes the listing shorter and easier to understand. Another advantage is, if we are to modify the function we need to make the changes at only one place in the program instead of four places.

Function Templates for User-defined Types

The **myMin()** function can even work on user-defined data types, like say provided the less-than operator (**<**) is overloaded in the **Date** class to compare **two** Date objects. We will have to also provide an overloaded **>>** operator function to output dates. (The working of such an overloaded operator function has already been discussed in [Chapter](#). The program given below illustrates this.

```
#include
using namespace std;

class Date
{
private:
    int day, mon, year;
public:
    Date (int d, int m, int y)
    {
        day = d;
        mon = m;
        year = y;
    }
```

```
int operator < (Date dt)
{
if (year < dt.year)
return 1;
if (year == dt.year && mon < dt.mon)
return 1;
if (year == dt.year && mon == dt.mon && day < dt.day)
return 1;

return 0;
}
friend ostream& operator << (ostream &o, Date &dt);
};

ostream& operator << (ostream &o, Date &dt)
{
o << dt.day << "/" << dt.mon << "/" << dt.year;
return o;
}
```

```
template < class T >
T myMin (T a, T b)
{
return (a < b) ? a : b;
}
```

```
int main()
{
```

```
int i = 10, j = 20;
cout << myMin (i, j) << endl;
Date dt1 (17, 11, 62), dt2 (23, 12, 65);
cout << myMin (dt1, dt2) << endl << endl;
return o;
}
```

One More Function Template

Let us write one more generic template function to help you fix your ideas. This one swaps contents of two variables.

```
#include
using namespace std;

class Date
{
private:
int day, mon, year;
public:
Date (int d = 0, int m = 0, int y = 0)
{
day = d;
mon = m;
year = y;
}
friend ostream& operator << (ostream &o, Date &dt);
};
ostream& operator << (ostream &o, Date &dt)
{
o << dt.day << "/" << dt.mon << "/" << dt.year;
```

```
return o;
}

template < class T >
void mySwap (T &a, T &b)
{
T c;
c = a;
a = b;

b = c;
}
int main()
{
int i = 10, j = 20;
mySwap (i, j);
cout << i << "\t" << j << endl;

char ch = 'A', dh = 'Z';
mySwap (ch, dh);
cout << ch << "\t" << dh << endl;

Date dt1 (17, 11, 62), dt2 (23, 12, 65);
mySwap (dt1, dt2);

cout << dt1 << "\t" << dt2 << endl;
return o;
```

}

The program defines a function template called `From` this template the compiler generates functions that will swap and and In the constructor of the **Date** class we have used default values for arguments. This is necessary to help us create objects of type **Date** in the **mySwap()** function, when it called for swapping contents of two **Date** objects.

Note that standard type conversions are not applied to function templates. When a call is encountered, compiler first looks at the existing instantiations for an “exact match” for the parameters supplied.

If this fails, it tries to create a new instantiation to create an ‘exact match’. If this fails, the compiler generates an error.

FUNCTION TEMPLATE OVERRIDE

What if we want a function to behave in one way for all data types except one? In such a case, we can override the function template for that specific type. For this we simply need to provide a non-templated function for that type as shown below.

```
void mySwap (double a, double b)
{
// some code here
}
```

This definition enables you to define a different function for **double** variables. Like other non-templated functions, standard type conversions (such as promoting a variable of type **float** to **double**) are now applicable.

Multiple Argument Types

We can write a function template that takes multiple types of arguments during one call. The following program shows such a function template.

```
#include
using namespace std;

template < class T, class S, class Z > void fun (T a, S b, Z
c)
{
    cout << a << endl << b << endl << c << endl;
}
int main()
{
    int i = 10;
    float j = 3.14f;
    char ch = 'A';
    fun (i, j, ch);
    return 0;
}
```

You must have noticed a small syntax variation in the function template of this program. We have put the template keyword and the function declarator on the same line:

```
template < class T, class S, class Z >
void fun (T a, S b, Z c)
```

This has got nothing to do with the multiple types of arguments that are being passed to the function template. We could as well have adopted the multi-line approach of earlier programs:

```
template < class T, class S, class Z >
void fun (T a, S b, Z c)
```

Templates versus Macros

In many ways, templates work like preprocessor macros, replacing the template argument with the given type. However, there are many differences between a macro like

```
# define min(i, j) ((i) < (j) ? (i) : (j))
```

and, a parallel function template template

```
< class T >
T min (T i, T j)
{
    return (i < j) ? i : j;
}
```

The macro given above also performs a simple text substitution and can thus work with any type. However it suffers from several limitations. These are as follows:

The macro is expanded without any type checking.

The type of the value returned isn't specified, so the compiler can't tell if we are assigning it to an incompatible variable.

In the macro the parameters **i** and **j** are evaluated twice. If either parameter has a post-incremented variable, the increment would take place twice.

When the macros are expanded by the preprocessor, the compiler error messages would refer to the expanded macro, rather than the macro definition itself. This makes bug hunting difficult.

To put it one line—macros are no match for templates.

A Template-based Sort

Let us now implement a template-based Selection sort function that can sort arrays of standard types as well as arrays of user-defined types. To achieve this we have defined a **Date** class containing an overloaded operator function to compare two Date objects with a view to determine which one is greater than the other. A **friend** function has also been implemented to output a **Date** object. Here is the program...

```
#include
using namespace std;

class Date
{
private:
    int day, mon, year;
public:
    Date (int d = 0, int m = 0, int y = 0)
    {
        day = d; mon = m; year = y;
    }
    int operator > (Date dt)
    {
```

```
if (year > dt.year)
return 1;
if (year == dt.year && mon > dt.mon)
return 1;
if (year == dt.year && mon == dt.mon && day > dt.day)
return 1;
return 0;
}
```

```
friend ostream& operator << (ostream &o, Date &dt);
};

ostream& operator << (ostream &o, Date &dt)
{
o << dt.day << "/" << dt.mon << "/" << dt.year;
return o;
}
```

```
template < class T >
void selectionSort (T a[], int sz)
{
T temp;

for (int i = 0; i < sz - 1; i++)
```

```
{
for (int j = i + 1; j < sz; j++)
{
```

```

if (a[i] > a[j])
{
temp = a[i]; a[i] = a[j]; a[j] = temp;
}
}
}
}

int main()
{
int arr[] = {-12, 23, 14, 0, 245, 78, 66, -9};
Date dtarr[] = {
Date (17, 11, 62), Date (23, 12, 65),
Date (12, 12, 78), Date (23, 10, 69)
};

int i;
selectionSort (arr, 8);
for (i = 0; i < 8; i++)
cout << arr[i] << endl;

cout << endl << endl;
selectionSort (dtarr, 4);
for (i = 0; i < 4; i++)
cout << dtarr[i] << endl;

return 0;

```

}

I do not intend to discuss here the actual working of the Selection Sort algorithm. This topic has been dealt with quite thoroughly in all the standard text-books on Data Structures. What you need to concentrate here is how to write function template that can work for standard as well as user-defined data types.

Class Templates

The concept of templates can be extended even to classes. Class templates are usually used for data storage classes. Such classes are known as container classes. Recall that, we implemented a **Stack** class to maintain a LIFO list in [Chapter](#). However, it could store data of only a single basic type, say an integer. If we were to store data of type **float** in a stack we would be required to define a completely new class. It follows that for every new data type that we wish to store, a new stack class would have to be created. Won't it be nice if we are able to write a single class specification that would work for all types. Enter **class**. Here is a program with class template in action.

```
#include  
using namespace std;  
  
const int MAX = 10;  
template < class T >  
class Stack  
{  
private:  
T stk[MAX];
```

```
int top;
public:
Stack()
{
top = -1;
}
void push (T data)
{
if (top == MAX - 1)

cout << "Stack is full" << endl;
else
{
top++;
stk[top] = data;
}
}
T pop()
{
if (top == -1)
{
cout << "Stack is empty" << endl;
return NULL;
}
else
{
T data = stk[top];
top--;
}
```

```
return data;
}
}
};

class Complex
{
private:
float real, imag; public:
Complex (float r = 0.0, float i = 0.0)
{
real = r;

imag = i;
}
friend ostream& operator << (ostream &o, Complex &c);
};

ostream& operator << (ostream &o, Complex &c)
{
o << "("<< c.real << ", "<< c.imag << ")";
return o;
}

int main()
{
Stack < int > s1;
s1.push (10);
```

```
s1.push (20);
s1.push (30);
cout << s1.pop() << endl;
cout << s1.pop() << endl;
cout << s1.pop() << endl;
```

```
Stack < float > s2;
s2.push (3.14f);
s2.push (6.28f);
s2.push (8.98f);
cout << s2.pop() << endl;
cout << s2.pop() << endl;
cout << s2.pop() << endl;
```

```
Complex c1 (1.5f, 2.5f), c2 (3.5f, 4.5f), c3 (1.2f, 0.6f);
```

```
Stack < Complex > s3;
```

```
s3.push (c1);
s3.push (c2);
```

```
s3.push (c3);
cout << s3.pop() << endl;
cout << s3.pop() << endl;
cout << s3.pop() << endl;
```

```
return o;
}
```

We have created three stacks here— **s2** and **s3** and pushed three values on each one. Then we have popped the values from the three stacks and displayed them on the screen.

Here's the output of the program...

```
30
20
10
8.98
6.28
3.14
(1.2, 0.6)
(3.5, 4.5)
(1.5, 2.5)
```

You can observe that the order in which the elements are popped from the stack is exactly reverse of the order in which they were pushed on the stack.

The way to build a class template is similar to the one used for building a function template. The **template** keyword and **< class T >** signal that the entire class will be a template.

```
template < class T >
class stack
{
// data and member functions using template argument T
```

```
};
```

The template argument **T** is then used at every place in the class specification where there is a reference to the type of the array. There are three such places—the definition of the argument type of the **push()** function, and the return type of the **pop()** function.

We have also declared a class called **Complex** and then pushed/ popped **Complex** objects to/from stack. This proves that we can create stacks of user-defined objects too from the class template. To be able to display the **Complex** objects through **cout** we have overloaded the **<<** operator.

We saw that in function templates the instantiation takes place when a function call is encountered. As against this, classes are instantiated by defining an object using the template arguments. For example,

```
Stack < int > s1;
```

creates an object, a stack that can store numbers of type **int**. The compiler reserves space in memory for this object's data, using type **int** wherever the template argument **T** appears in the class specification. It also reserves space for the member functions (if these have not already been placed in memory by

another object of type **Stack < int** These member functions also operate exclusively on type

When we create a **Stack** object that stores objects of a different type, say space is now created for data, as well as a new set of member functions that operate on type

As with normal classes can we not define the member functions of a class template outside the class? We can, but it needs a different syntax as shown below.

```
template < class T >
void Stack < T > :: push (T data)
{
if (top == MAX - 1)
cout << endl << "Stack is full";
else
{
top++;
stk[top] = data;
}
}
```

Note that the expression **template < class T >** must precede not only the class definition, but each externally defined

member function as well. The name **Stack < T >** is used to identify the class of which **push()** is a member.

A Linked List Class Template

Let us now implement a general-purpose linked list class using templates. Using this class template we can easily maintain a linked list of integers or a linked list of floats, or a linked list of user-defined data type called **Employee**. Let us assume that the user-defined data type stores information about name, age and salary of an employee. The **Employee** class would need two member functions—a constructor and an overloaded << operator.

I am giving below the skeleton program that implements the class template to maintain two linked lists—one for integers and another for employee data. You can try to implement the functions of **LinkedList** class and **Employee** class.

```
#include  
#include  
using namespace std;  
  
class Employee  
{  
private:
```

```
string name; int age; float sal;
public:
Employee (string n = "", int a = 0, float s = 0.0);
friend ostream& operator << (ostream& s, emp& e);
};

template < class T >
class LinkList
{
private:
struct node
{

};

T data;
node *link;
} *p;
public:
LinkList ();
~ LinkList ();
void append (T);
void addAtBeg (T);
void addAfter (int, T);
void del (int);
void display();
int count();
};

int main()
{
```

```
LinkList < int > l1;
cout << "No. of elements in Linked List =" << l1.count() <<
endl;
l1.append (11); l1.append (22); l1.append (33);
l1.append (44); l1.append (55); l1.append (66);
l1.addAtBeg (100); l1.addAtBeg (200);
l1.addAfter (3, 333); l1.addAfter (6, 444);
l1.display();
cout << "No. of elements in linked list =" << l1.count() <<
endl;
l1.del (200); l1.del (66);
l1.del (0); l1.del (333);
l1.display();
```

```
cout << "No. of elements in linked list =" << l1.count() <<
endl;
```

```
LinkList < Employee > l2;
cout << "No. of elements in Linked List =" << l2.count() <<
endl;
Employee e1 ("Sanjay", 23, 1100.00);
Employee e2 ("Rahul", 33, 3500.00);
Employee e3 ("Rakesh", 24, 2400.00);
Employee e4 ("Sanket", 25, 2500.00);
Employee e5 ("Sandeep", 26, 2600.00); l2.append (e1);
l2.append (e2);
l2.append (e3);
l2.append (e4);
```

```
l2.append (e5);
l2.display();
l2.del (3);
l2.display();
cout << "No. of elements in linked list =" << l2.count() <<
endl;
l2.addAtBeg (e5);
l2.display();
l2.addAfter (3, e1);
l2.display();
cout << "No. of elements in linked list =" << l2.count() <<
endl;

return o;
}
```

Tips about Templates

Let me now give a few tips that you would find useful.

While declaring a function template or a class template we can replace the keyword **class** with the keyword **typename** as shown below.

```
template < typename T >
void selectionSort (T a[], int sz)
{
// code
}
```

```
template < typename T >
class Stack
{
// code
};
```

The name of the template class (say is expressed differently in different contexts. Within the class specification, it is simply the name, as in

```
class Stack {};
```

For externally defined member functions, it is the class name plus the template argument name as in

```
void Stack < T > :: push (T data) {}
```

Lastly, when you define actual objects for storing a specific data type, it is the class name plus this specific type as in

```
Stack < float > s1; // object of type Stack
```

You must exercise considerable care to use the correct form in the correct context. It's easy to forget to add the `< T >` or `< float >` to the The compiler hates it when you get it wrong.

Be careful about the syntax when a member function returns a value of its own class. Suppose we define a class template called If a member function `fun()` of this class returns a type and we have to define this function outside the template class we need to use `Sample < T >` for the return type as well as preceding the scope resolution operator as shown below.

```
Sample < T > Sample < T > :: fun (Sample s)
{
}
```

The class name used as a type of a function argument, on the other hand, doesn't need to include the `< T >` designation.

Template arguments can take default values. The values of these arguments then become compile-time constants for that particular instantiation of the template. For example,

```
template < class T, int max = 50 >
class Stack
{
private:
T arr[max];
};
```

We can use this class template through statements like

```
Stack < int, 10 > s1; // can store 10 integers in the stack
Stack < float > s2; // can store 50 floats in the stack
```

As in case of functions with default values for arguments, here too, the default values can be given to the trailing

arguments of the template's argument list.

We can inherit a new template from an existing one. For example,

```
template < class T >
class NewSample: public Sample < T >
{
};
```

Every time we instantiate a template the code in the template is generated anew. If some of the functionality of a template does not depend on type, it can be put in a common base class to prevent unnecessary reproduction of that code.

Templates should be used while creating a type-safe collection class that can operate on data of any type.

Variadic Templates

A **variadic** template is a function or class template that supports an arbitrary number of arguments. The following program shows how to use a variadic function template to get a sum of an arbitrary number of arguments passed to it.

```
#include
using namespace std;

template < class T >
double getSum (T t)
{
    return t;
}

template < class T, class... S >
double getSum (T first, S ... rest)
{
    return first + getSum (rest...);
}

int main()
```

```

{
double s1 = getSum (10, 20, 30);
double s2 = getSum (1.5, 2.5, 3.5, 4.5);
double s3 = getSum (10, 2.5, 20, 3.5);

cout << s1 << "\t" << s2 << "\t" << s3 << endl;
}

```

In **main()** we have called **getSum()** with arbitrary number of arguments. Each time we expect **getSum()** to return a double that represents the sum of arguments passed to it.

Interestingly, we have two implementations of **getSum()** template function. The first one receives only one argument, whereas the second receives variable number of arguments. Note the use of ellipsis (...) and their placement in the second implementation.

```

template < class T, class... S >
double getSum (T first, S... rest)
{
    return first + getSum (rest...);
}

```

An ellipsis is used in different ways by variadic template function. These are as follows:

class... **S** represents multiple types, which you can mentally think of as **S₁**, **S₂**, **S₃**, etc. **S** is often called a parameter pack, indicating that it represents multiple types.

S... **rest** indicates that the function **getSum()** is going to receive multiple argument of different types, apart from the **first** argument of type

rest... indicates that **getSum()** is going to get called recursively by passing to it the values contained in

Let us now see how the recursive calls would work. The first version of **getSum()** represents the base case and the second version represents the recursive case. The base case accepts one argument whereas the recursive case accepts one or more arguments **T** and

The call **getSum (10, 20, 30)** expands as shown below.

```
getSum (10, 20, 30);
10 + getSum (20, 30);
10 + (20 + getSum (30));
10 + (20 + getSum (30));
```

In the first three calls the recursive version of **getSum()** gets called, whereas, in the last call the base version gets called.

Applications of Templates

We now know that a function/class template represents a family of functions/classes. Templates are of great utility to C++ programmers. They are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The C++ Standard Library provides many useful template classes. These include:

Smart pointer classes that are of great help in avoiding memory leaks and dangling pointers.

Classes in iostream library that let us carry out I/O in C++.

In addition to this, Standard Template Library (STL) contains many useful classes to manage collections of different types efficiently and elegantly. The STL library is discussed in detail in [Chapter](#)

Exercise

State whether the following statements are True or False:

We can inherit a new class from a class template.

If there is a function template called **max()** then a specific version of it would be created when **max()** is called with a new type.

The compiler generates only one version of function template for each data type irrespective of the number of calls that are made for that type.

Using templates saves memory.

We can override a function template for a particular type.

A function template can have multiple argument types.

Templates are type safe, whereas, macros are not.

Class templates are usually used for container classes.

A class template member function can be defined outside the class template.

Template arguments can take default values.

While defining a function template we should use template < class T >, whereas while defining a class template we should use template < typename T >.

The template parameter (usually denoted as T, S, Z, etc.) of a function template can be used to specify the types of the arguments to the function, to specify the return type of the function and to declare variables within the function.

Template parameter names among template definitions must be unique.

Templates enable us to specify, with a single code segment, an entire range of related overloaded functions called, or an entire range of related classes.

The related functions generated from a function template all have the same name.

Answer the following:

Write a function template to check the template class **LinkedList** discussed in the text. Invoke this function template from

Write a program that will implement a binary tree as a class template.

Write a program to implement a doubly linked list as a class template.

Write a variadic template function that prints the variable number of parameters passed to it.

Write a variadic template function that evaluates the following

$x + + + ^2 + \dots$

KanNotes

Templates promote source-code level reuse, whereas Inheritance and Containership promote object-code level reuse

It is possible to create template functions as well as template classes

Once the template function / class is ready we can use them with any standard or user-defined type

Syntax of defining and calling a template function:

```
// template function definition
template < class T > // or template < typename T >
void printArray (T[] arr)
{
..
}
```

```
// call to template function
int intarr[] = {10, -2, 37, 42, 15};
printArray (intarr);
```

Template function can receive multiple types as in

```
template < class T, class S, class Z >
void printTypes (T a, S b, Z c)
{
..
}
```

Syntax for using and defining a template class:

```
// using template class
stack < int > s1;
s1.push (10);
```

```
// defining template class
template < class T >
class Stack
{
..
}
```

It is possible to write a function or a class variadic template that supports an arbitrary number of arguments

Exception Handling

Writing programs that work with ideal input and ideal environment is not enough. We have to anticipate all possibilities where things may go wrong and incorporate measures in our program to deal with them. This chapter explains how to create such robust programs using C++ exception handling mechanism.

Exception Handling in C++

Using Ready-made Exception Class

Using Ready-made Exceptions in Implementing Queue

One More Example

Using User-defined Exception Class

A Few Tips

Exception Specification

Unhandled Exceptions

Smart Pointers and Dynamic Containers

Exercise

KanNotes

Programming is a difficult art. No matter how much confidence you have in your programming ability, several things may go wrong during its development. This includes typing errors, compilation errors, linking errors runtime errors. The first three types of errors are comparatively easier to tackle. But when errors occur during execution, our program has to deal with the situation in an elegant fashion. This chapter discusses how to deal with such types of errors.

The errors that occur at runtime—i.e., during execution of the program—are called Exceptions. The reasons why exceptions occur are numerous. Some of the more common ones are as follows:

Falling short of memory

Inability to open a file

Exceeding the bounds of an array

Attempting to initialize an object to an impossible value

Division by zero

Stack overflow

Arithmetic over flow or under flow

Attempt to use an unassigned reference

Unable to connect to Server

When such exceptions occur, the programmer has to decide a strategy according to which he would handle the exceptions. The strategies could be, displaying error messages on the screen, or displaying a dialog box in case of a GUI environment, or requesting the user to supply better data, or logging the error message in a file or simply terminating the program execution.

Exception Handling in C++

C++ provides a systematic, object-oriented approach to handling runtime errors. The exception handling mechanism of C++ uses three keywords— and Let us understand their purpose.

Suppose during the course of execution of a function (either global or class member) an error occurs. We need to report this by performing following two steps:

Creating an object called exception object and storing information about the exceptional condition in it.

Throwing the exception object using the keyword

The place from where an exception object is thrown is called throw point. We can create an exception object from ready-made exception classes provided in C++ standard library. Each of these exception classes represents a different exception situation. For example, one of exception class represents an arithmetic overflow situation, another represents result of

arithmetic going out of range, etc. All these classes are derived from a base class called

[Figure 11-1](#) shows the hierarchy of exception classes available in standard library.

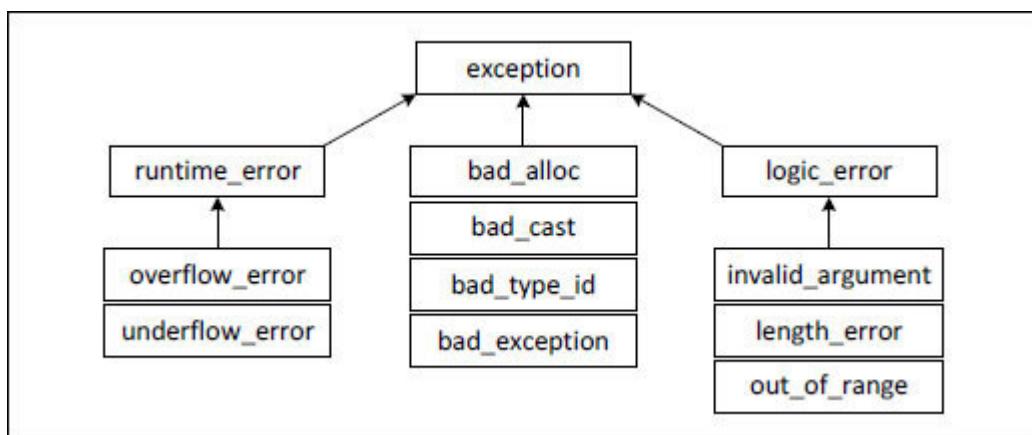


Figure 11-1. Hierarchy of C++ standard library exception classes.

If we anticipate an exceptional situation other than the ones represented through these exception classes, then we can define our own exception class, create an object from this class and then throw that object to report the error situation. To begin with, we would concentrate on ready-made exception classes and later move to user-defined exceptions.

The code in the application that anticipates an exception to occur during its execution is enclosed in a **try** When the

exception occurs and an exception object is thrown, the control is transferred to another section of code in the application called **exception handler** or a **catch block**. Thus, runtime errors generated in the **try** block are caught in the **catch** block. The passing of control (often called propagation) from the place where the exception occurred in the **try** block to the place where it is handled (i.e. the catch block) is done by C++ runtime.

Note that the code that doesn't expect an exception to occur need not be present within the **try** block.

The following code snippet shows the organization of **try** and **catch** blocks. It is not a working program, but it clearly shows how and where the various elements of the exception mechanism are placed.

```
class Sample
{
public:
void fun()
{
// some code
if (some error occurs)

{
// throw exception from this throw point
```

```
// it would be caught in the catch block throw overflow_error
(..);
}
}
};

int main()
{
// try block
try
{
Sample s;
s.fun();
}
catch (overflow_error &e) // exception handler or catch block
{
// code to do something about the caught exception
}
return 0;
}
```

Here **Sample::fun()** is the function in which an exception may occur during execution. Hence, the call to **fun()** has been placed in the **try** block. If overflow error doesn't occur then **fun()** will execute normally and control will return from it. If an exception indeed occurs during execution of **fun()** then an

exception object of class **overflow_error** will be created and thrown. If there are statements below the throw point, or below the call to **fun()** in they would not executed, since once an exception object is thrown the **try** block expires.

The thrown exception object is caught in the **catch** block that immediately follows the **try** block. There are two things that our code can do on catching a thrown object. These are as follows:

Perform a graceful exit

Rectify the situation that led to the exceptional condition and continue the execution

These two possibilities are depicted in [Figure](#)

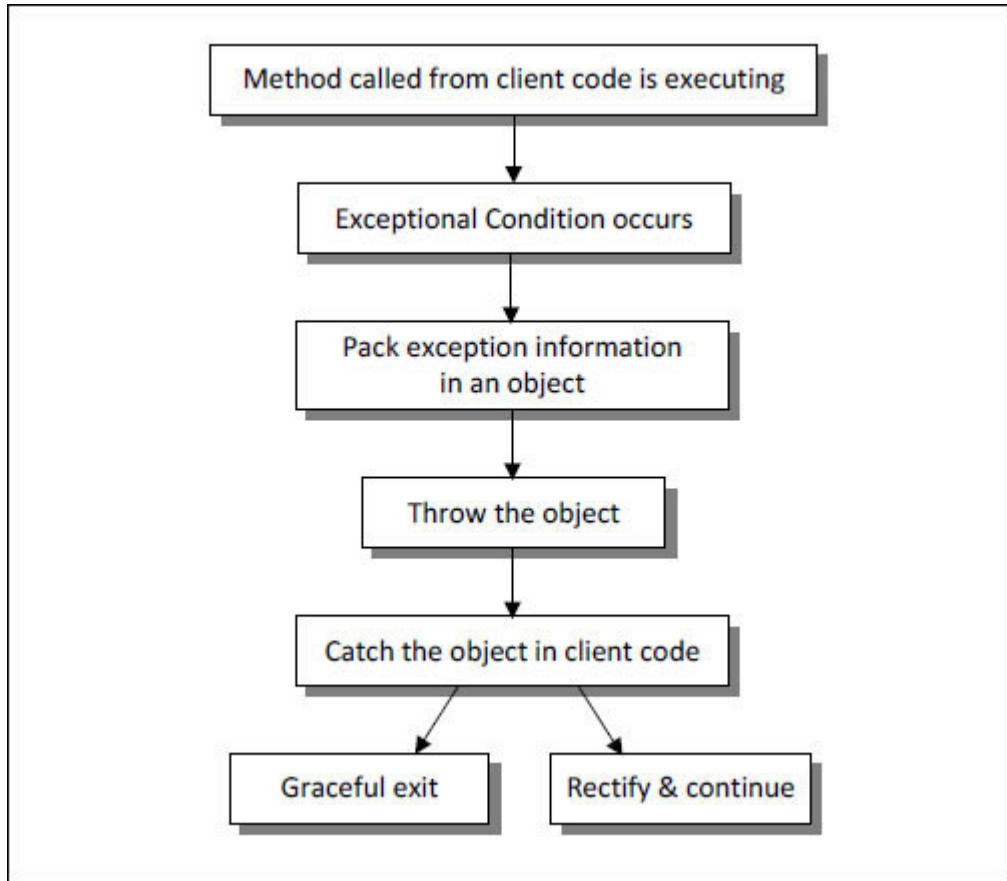


Figure 11-2. C++ exception handling mechanism.

Once the catch block has been executed, control passes to the statements below the catch block, unless there is a **return** or **exit()** in the catch block.

Using Ready-made Exception Classes

As shown in [Figure](#) all standard library exception classes are inherited from **exception** class. This class has been declared in the header file ‘exception’, whereas, **logic_error** and **runtime_error** have been declared in ‘stdexcept’. The list of classes in [Figure 11-1](#) is not complete and there are other exception classes related with errors in memory allocation, usage of smart pointers, filesystem etc.

Let us now write a program that makes use of some of these exception classes. Here is the program...

```
#include  
#include  
#include  
#include  
using namespace std;  
  
int main()  
{  
try  
{
```

```

ifstream infile ("Sample.cpp");
if (!infile)
throw runtime_error ("File opening failed");
string str1 ("Yankee");
string str2 ("Doodle");
str1.append (str2, 7, 3);
cout << str1 << endl;
}
catch (logic_error &le)
{
    cout << le.what();
}
catch (runtime_error &rte)
{
    cout << rte.what();
}

```

This program attempts two things in the **try** block—open a file for reading and append a part of string **str2** at the end of string If the file is opened successfully, program proceeds to append strings. If not, then a **runtime_error** exception object is created and thrown. Once the object is thrown, the **try** block is terminated. This thrown object is caught in the second **catch** block and the message ‘File opening failed’ stored in the **runtime_error** object is displayed on the screen. Since the

thrown object is caught by reference, another copy of this object is not created.

If control reaches the appending logic then we attempt to extract from ‘Doodle’ 3 characters starting at character and append them to ‘Yankee’. Since ‘Doodle’ has only 6 characters there is no question of extracting and 9th character. Hence the **append()** function throws a **out_of_range** exception. This exception is caught by the first **catch** block. Note that we have caught it using a **logic_error** reference and not **out_of_range** reference. This is possible because **logic_error** is the base class of Thus, **le** is an upcasted reference. Using it we call the **what()** method to print an error message ‘invalid string position’. From where does this message come? Well, when append throws the **out_of_range** object, it stores this message in the exception object, which is then retrieved using

Let us now understand the difference between **logic_error** and Some errors are detectable before the program executes. For example, division operation in which denominator is 0. When such exceptions occur, **logic_error** object should be thrown. As against this, there are some errors that can get detected only when the program executes. For example, opening a file for reading. In such cases, **runtime_error** exceptions should be thrown.

USING READY-MADE EXCEPTIONS IN IMPLEMENTING QUEUE

Let us now put exception handling mechanism to work in a program that maintains a queue data structure (First In, First Out list). We would use exception handling to report errors in two situations. These are as follows:

When the program attempts to store more objects in the queue than what it can accommodate.

When the program tries to remove an object from the empty queue.

Here is the program...

```
#include  
#include  
using namespace std;  
  
#define MAX 4  
class Queue  
{  
private:  
    int arr[MAX];  
  
    int front, rear;
```

```
public:  
Queue()  
{  
front = -1;  
rear = -1;  
}  
void addQ (int item)  
{  
if (rear == MAX - 1)  
throw exception ("Queue is full");  
rear++;  
arr[rear] = item;  
if (front == -1)  
front = 0;  
}  
int delQ()  
{  
int data;  
if (front == -1)  
throw exception ("Queue is empty");  
data = arr[front];  
if (front == rear)  
front = rear = -1;  
else  
front++;  
return data;  
}  
};
```

```
int main()
{
    Queue q;
    try
    {
        q.addQ (11);
        q.addQ (12);
        q.addQ (13);
        q.addQ (14);
        ddQ (15); // oops, queue is full
        int i;
        i = q.delQ();
        cout << "Item deleted =" << i << endl;
        i = q.delQ();
        cout << "Item deleted =" << i << endl;
        i = q.delQ();
        cout << "Item deleted =" << i << endl;
        i = q.delQ();
        cout << "Item deleted =" << i << endl;
        i = q.delQ(); // oops, queue is empty
        cout << "Item deleted =" << i << endl;
    }
    catch (exception &err)
    {
        cout << endl << err.what() << endl;
    }
    return 0;
}
```

}

In our program we have a **Queue** class that contains methods **addQ()** and **delQ()** for addition and deletion of elements to/from a queue, respectively. The **front** and **rear** data members are used to keep track of the front end of the queue and the rear end of the queue, respectively.

While executing **addQ()** there is a possibility that the array in which we store the queue elements is full, and cannot take any new elements. Similarly, while executing **delQ()** there is a possibility that we are removing an element from the queue, when it is already empty. Both of these are exceptional situations.

Since exceptions may occur while executing **addQ()** and calls to them have been enclosed in **try** block in We have purposefully kept the capacity of the queue small so that it's easier to trigger an exception while adding/deleting items from/to queue.

When the exception occurs, we have created a new **exception** object by passing to its constructor a relevant message. To be able to use the standard library **exception** class, we have included the header file 'exception'. Once the **exception** object is created we have used the **throw** keyword to throw it.

This thrown **exception** object is caught in the catch block by reference. As a result, a copy of the thrown object is not created, thereby avoiding duplication and saving space. Now that the control has shifted to the catch block (try block expires when first exception occurs), statements in it are executed. We have simply printed the message stored in the exception object by calling its **what()** method.

Similar set of actions will take place when we attempt to delete an element from an empty queue. To be able to experience that, you can comment out a few calls to

Let's summarize the events that we have discussed above:

Code is executing normally outside a **try** block.

Control enters the **try** block.

A statement in the **try** block causes an error in a member function.

The member function creates and exception object

The member function throws the exception object.

Control transfers to the exception handler block) following the **try** block.

The exception is processed in the catch block.

You can appreciate how clean this procedure is—it lets you separate the code that detects exceptions from the code that handles it. As a result, exception thrown from same piece of code can be handled differently by different clients by writing appropriate code in their respective catch blocks.

ONE MORE EXAMPLE

When we allocate memory using new there is a possibility that the amount of memory that we are trying to allocate is not available. In such a case, the **new** operator function throws an exception called We can catch this exception as shown in the following program.

```
#include  
#include  
using namespace std;  
  
int main()
```

```
{  
long *p[100];  
  
try  
{  
  
for (int i = 0; i < 100; i++)  
{  
p[i] = new long[50000000];  
cout << i << "Array of longs allocated" << endl;  
}  
}  
catch (bad_alloc &e)  
{  
cout << "Memory allocation exception:" << e.what() << endl;  
}  
}
```

Here we are trying to allocate 100 arrays, each holding 50,000,000 long ints. When I executed the program, after allocating 6 such arrays, an exception occurred after iteration. Based on RAM capacity on your machine and the available disk space for virtual memory you may get a different result.

We have included the header file ‘new’ since it contains the declaration of **bad_alloc** exception class. The exception thrown

has been caught and reported, along with the exception type returned by the **what()** method.

Instead of this this, if we so desire, we can set up our own handler to which the control would be passed when memory allocation exception occurs. This is shown in the following program.

```
#include  
#include  
using namespace std;  
  
void my_allocandler()  
  
{  
    cout << "Memory allocation exception" << endl;  
    abort();  
}  
int main()  
{  
    long *p[100];  
  
    set_new_handler (my_allocandler);  
    for (int i = 0; i < 100; i++)  
    {  
        p[i] = new long[50000000];  
        cout << i << "Array of longs allocated" << endl;
```

```
}
```

```
}
```

Using the **set_new_handler()** declared in ‘new’ header file we can set up **my_allocandler()** as the function to be called when memory allocation failure occurs. This user-defined handler function should not receive any value and should not return any value. In this handler function we have merely displayed an error message and terminated the program by calling

Using User-defined Exception Class

In the program in the previous section when an exception occurs we merely get a message printed out. At times this is not enough. We should get some additional information about the context in which the exception occurred. For this we need to pack more information into the exception object being thrown. This can be done by creating our own exception class. This is shown in the following program:

```
#include  
#include  
#include  
#include  
using namespace std;  
  
#define MAX 4  
class QueueException: public exception  
{  
private:  
    string errMsg;  
public:  
    QueueException (string str, int front, int rear)  
{
```

```
ostrstream s;
s << str << endl << "front =" << front << endl
<< "rear =" << rear << ends; errMsg = s.str();
}
string what()
{
return errMsg;
}

};
```

```
class Queue
{
private:
int arr[MAX];
int front, rear;
public:
Queue()
{
front = rear = -1;
}
void addQ (int item)
{
if (rear == MAX - 1)
throw QueueException ("Queue is full", front, rear);
rear++;
arr[rear] = item;
if (front == -1)
```

```
front = 0;
}

int delQ()
{
    int data;
    if (front == -1)
        throw QueueException ("Queue is empty", front, rear);
    data = arr[front];
    if (front == rear)
        front = rear = -1;
    else

        front++;
    return data;
}

};

int main()
{
    Queue q;

    try
    {
        q.addQ (11);
        q.addQ (12);
        q.addQ (13);
        q.addQ (14);
```

```

ddQ (15); // oops, queue is full
int i;
i = q.delQ();
cout << "Item deleted =" << i << endl;
i = q.delQ();
cout << "Item deleted =" << i << endl;
i = q.delQ();
cout << "Item deleted =" << i << endl;
i = q.delQ();
cout << "Item deleted =" << i << endl;
i = q.delQ(); // oops, queue is empty
cout << "Item deleted =" << i << endl;
}
catch (QueueException &err)
{
cerr << endl << err.what() << endl;

}

return o;
}

```

Here we have created our own exception class and inherited it from the standard library **exception** class. When an exception occurs and the exception object is to be constructed, we have called the constructor of **QueueException** class and passed to it the error string, **front** and To build one string out of these, the constructor has output them to an **ostream** object. To mark the end of string **ends** is used, rather than the usual

When **what()** method is called from the catch block the string is extracted from the **ostrstream** object and returned. This string is then sent to **cerr** instead of **cout**. This is considered to be a good practice, as it lets us to redirect the **cerr** (standard error) stream to a file, if desired.

A Few Tips

Now that we have written a few programs that use exception handling a few tips are in order.

It is not necessary that the statement that causes an exception be located directly in the **try** block. It may as well be present in a function that is being called from the **try** block.

try blocks can be nested. When an exception is thrown, if inner **try** block doesn't have a corresponding **catch** block, then the outer **try** block's **catch** handlers are inspected for a match. This is known as stack unwinding.

There can be more than one exception handler for one **try** block, as shown below.

```
try
{
// code that may generate exceptions
}
catch (exceptiontype1 id1)
{
```

```
// handle exceptions of type1  
}  
catch (exceptiontype2 id2)  
{  
// handle exceptions of type2  
}
```

This is a good idea as it helps you to distinguish between types of exceptions.

When an exception is thrown, the **catch** blocks are examined in the order in which they are written. When it finds a match, the exception is considered handled and no further searching takes place.

Don't catch every exception in an **exception** object. This would work since **exception** is the base class from which other exception classes are usually derived. More granular the catch block, more pin-pointing would the exception handling be.

Do not place a **catch** that catches a base-class object before a **catch** that catches a derived class object. If you do, then the base-class **catch** will catch all objects of classes derived from that base class. As a result, derived-class **catch** will never execute.

You can write a catch-all block using the syntax

```
catch (...)  
{  
}
```

You should use this as the last catch block in a series of catch blocks. It is often called a default catch block.

It is a bad programming practice to write a catch-all block and do nothing in it. This is like catching an exception and ignoring it.

Don't use exception handling for cosmetic purpose. It should serve the purpose of rectifying the exceptional situation or perform a graceful exit.

If we are writing a class library for somebody else to use, we should anticipate what could cause problems to the program using it. At all such places we should throw exceptions.

If we are writing a program that uses a class library, we should provide try and **catch** blocks for any exceptions that the library may throw.

Exceptions impose an overhead in terms of program size and (when an exception occurs) in time. So we should not try to overuse it. Make it optimally elaborate—not too much, not too little.

Exception Specification

A function can advertise which exceptions it may throw. This is known as exception specification or a throw-list. Once this list is specified the user of that function can catch them through one or more catch blocks. Once specified, the function cannot throw any other exception other than the ones mentioned in the list. Here are a few examples...

```
// can throw any exception
void closeAccount();
```

```
// cannot throw any exception
void openAccount () throw ();
void openAccount() noexcept; // C++ 17 specific
```

```
// may throw standard exception
void fun() throw (overflow_error);
```

```
// may throw user-defined exceptions
bool withdraw (int no) throw (InsufficientFundsExc,
InvalidAccnoExc);
```

Note that a function can either throw exception objects of exception classes mentioned in the throw-list, or objects created from their publicly derived classes.

Unhandled Exceptions

If the function throws an exception other than those mentioned in the throw-list, then a pre-defined function called **unexpected()** gets called. This function calls **terminate()** function, which in turn calls **abort()** function to terminate the program execution.

If we wish that our version of **unexpected()** or **terminate()** function be called instead of the pre-defined ones, we need to set them up as shown in the following program:

```
#include
#include
using namespace std;
class Exception1 {};
class Exception2 {};
class Exception3 {};
void my_terminate()
{
    cout << "Call to my terminate" << endl;
    abort();
}
void my_unexpected()
```

```
{  
cout << "Call to my_unexpected" << endl;  
throw; // rethrow Exception3  
}  
void fun() throw (Exception1, Exception2)  
{  
throw Exception3();  
}  
int main()
```

```
{  
set_unexpected (my_unexpected);  
set_terminate (my_terminate);  
  
try  
{  
cout << "In try block" << endl;  
fun();  
}  
catch (Exception1 e)  
{  
cout << "Caught Exception1" << endl;  
}  
catch (Exception2 e)  
{  
cout << "Caught Exception2" << endl;  
}  
}
```

When `fun()` throws `Exception3` it cannot be caught since there is no matching catch block. As a result, `unexecuted()` should get called. But we have provided our version of this function by using So control lands into Here we have re-thrown `Exception3` by using an empty `throw` statement. This time `my_terminate()` gets called, as we have set this up using This function merely prints a message and terminates execution be calling

Note that the prototypes of our handler function should be as follows:

```
void my_unexpected(); // receives nothing, returns nothing
void my_terminate(); // receives nothing, returns nothing
```

Smart Pointers and Dynamic Containers

While creating and destroying objects dynamically, a common set of problems are repeatedly faced. These are discussed below.

Memory This happens when we allocate memory but forget to de-allocate it. For example, in the code snippet given below when control returns from memory would leak as p would die, but object it was pointing to would continue to remain allocated and we would have no way to reach it.

```
void fun()
{
    Sample *p;
    p = new Sample; // create object
    p -> show(); // use object
}
```

Dangling If two pointers are pointing to the same object and one of the pointers is used to delete the object, the object would be destroyed, but the second pointer would continue to

point to it. For example, in the following code snippet **p2** would dangle when control returns from

```
void fun()
{
    Sample *p1, *p2;
    p1 = new Sample; // create object
    p2 = p1; // copy pointer
    p1 -> show(); // use object
    delete p1; // delete object pointed to by p1
}
```

Deleting an object If we delete a dynamically allocated object twice, the second delete will cause an undefined behavior. Anything can happen—from nothing to program crash.

```
void fun()
{
    Sample *p;
    p = new Sample; // create object
    p -> show(); // use object
    delete p; // delete object
    delete p; // attempt to delete object - undefined behaviour
}
```

Wrong deletion of an When we allocate an array using `new` we must always delete it using `delete[]`. In the following code snippet we have used `delete` instead of `delete[]`. As a result, the destructor would be called only for the first object in the array, causing memory leak.

```
void fun()
{
    Sample *p;
    p = new Sample[10]; // create 10 objects, call Ctor 10 times
    p[0] -> show(); // use 0th object
    p[1] -> show(); // use 1st object
    delete p; // deletes only first object, calls destructor only once
}
```

To avoid these problems caused by raw pointers that we saw above it has been suggested that `new` and `delete[]` operators should never be used. Instead, we should use one of the following entities:

Smart pointers like `unique_ptr< T >`, `shared_ptr< T >` and `weak_ptr< T >`

Dynamic containers like `set< T >`, `vector< T >` etc.

Both of them are available in the form of class templates in C++ standard library.

Smart pointers mimic a raw pointer by providing overloaded * and -> operators in it. At the same time, when we create an object using it, it is guaranteed to get destroyed properly without causing a memory leak.

If we use dynamic containers then we are not required to use the smart pointers **weak_ptr** explicitly. Naturally, using dynamic containers is the preferred way of acquiring and deleting resources in modern C++ programming. Hence, we would not go into the details of smart pointer solution and learn dynamic containers in next chapter.

Exercise

[A] State True or False:

The exception handling mechanism is supposed to handle compile time errors.

It is necessary to declare the exception class within the class in which an exception is going to be thrown.

Every thrown exception must be caught.

For one **try** block there can be multiple **catch** blocks.

The **catch** block and the exception handler are one and the same thing.

When an exception is thrown an exception class's constructor gets called.

try blocks cannot be nested.

Proper destruction of an object is guaranteed by exception handling mechanism.

[B] Answer the following:

Implement an exception handling mechanism that reports stack full and stack empty exceptions for a **Stack** class.

Create a **Customer** class containing **accountno** and **balance** as its data members. Create 5 objects of this class. Implement a **withdraw()** method that helps withdraw amount from a particular account. This method should throw an exception called **BankException** if the amount being withdrawn makes the balance in an account go below Rs. 1000. In the **catch** block report the customer details for the failed transaction.

KanNotes

While creating and executing a C++ program things may go wrong at 3 different stages:

During Compilation: Reported by - Compiler, Action - Rectify program

During Linking: Reported by - Linker, Action - Proper #include statements

During Execution (runtime): Reported by - C++ Runtime, Action - Tackle it on the fly

Examples of Runtime errors:

Memory Related - Stack / Heap overflow, Exceeding the bounds of an array

Arithmetic Related - Divide by zero, Arithmetic over flow or under flow

Others - Attempt to use an unassigned reference, File not found

When a method called from client code is executing an Exceptional Condition may occur. When this condition occurs two things should be done

Pack exception information in an object

Throw the exception object

Two things that can be done when the exception object is thrown:

Catch the object in client code

Throw it further

If we throw the exception object further - Default exception handler Catches the object & terminates

If we catch the exception object in client code we can either perform a Graceful exit or Rectify the exceptional situation & continue

Z Ways to create Exceptional Condition objects

From C++ standard library exception classes

From User-defined exception classes

Advantage of tackling exceptions in OO manner:

- More info can be packed into Exception objects
- Propagation of exception objects to caller is managed by C++ Runtime

How C++ facilitates OO exception handling:

- By providing keywords - try, catch, finally, throw, throws
- By providing ready-made exception classes
- Advertise - Let methods advertise possibility of an exception

How to use try - catch

try block - Enclose in it the code that you anticipate would cause an exception

catch block - Catch the thrown exception in it. It must immediately follow the try block

When exception is thrown control goes to catch block. Once catch block is executed, control goes to the next line after catch block(s), unless there is a return or throw in the catch block

try block:

- Can be nested inside the another try block
- If inner try doesn't have a catch, outer try's catch handlers are inspected for a match

catch block:

- Multiple catch blocks for one try block are OK
- At a time only one catch block goes to work

- Order of catch blocks is important - Derived to Base

Exception handling tips:

- Don't catch & ignore an exception
- Don't catch everything using 'exception', distinguish between types of exceptions
- Make it optimally elaborate - Not too much, not too little

Use Smart Pointers or Dynamic Containers instead of new, new[], delete and delete[]

Standard Template Library

Apart from the inherent features of OOP, it is the powerful Standard Template Library (STL) that makes C++ so popular with programmers who are looking to write robust and efficient programs. This chapter discusses the key components of STL.

Standard Template Library

Components of STL

Containers

Iterators

Algorithms

vector Container

Other Operations

Vector of Point Objects

list Container

Sets and Multi-sets

Maps and Multi-maps

stack Container

queue Container

Function Object

Exercise

KanNotes

The Standard Template Library (STL) consists of reusable classes that implement many common data structures. Initially, STL was a separate library, but has now been merged with C++ standard library. Since then, though the industry continues to use the acronym STL, the C++ standard document refers to it only as standard library. As now it is a part of standard library, to use it, we do not have to download it separately. This chapter discusses the various components of STL and shows how to use them in C++ programs.

Standard Template Library

STL has been developed by Alexander Stepanov and Meng Lee at the Hewlett Packard laboratories in Palo Alto, California.

While creating many C++ programs we need some or the other data structure to handle the data. Instead of creating and maintaining these data structures ourselves, we can use the ready-made STL classes to do that. This approach has following advantages:

Implementing each data structure like linked list, priority queue, map, etc. requires substantial work.

Many data structures need complex usage of pointers. A small omission or oversight may cause memory-access violations and memory-leaks.

By using STL we can concentrate on the problem that we are trying to solve and leave it to STL to handle the data.

Suppose in a large project programmers working on for different tasks need the same data structure. If they decide to create classes to handle this data structures individually, the

code will become difficult to modify, maintain and debug. Moreover, this approach encourages reinvention rather than reuse.

Thus using STL promotes reuse and can save substantial development time, energy and money. It also reduces testing and debugging time.

STL was conceived and designed for performance and flexibility. Moreover, all the classes in STL are template-based. As a result, they can be reused to work with standard as well as user-defined types. This extensibility of STL is another reason for STL's popularity amongst programmers.

Components of STL

STL has three key components—Containers, Iterators and Algorithms. Of these, STL containers are data structures capable of storing objects of almost any data type. STL iterators are used to manipulate the elements of a STL-container. Lastly, STL algorithms are used to process the data structures.

For example, a STL container called **vector** can be used to store several integers in the form of an array. A STL iterator can be used to traverse through this array in forward or reverse direction; and a STL algorithm called **sort()** can be used to sort the elements of the vector.

Note that manipulating containers with iterators not only offers convenience, but when combined with algorithms, makes the program expressive, flexible and efficient. Let us now take a closer look at the different STL components.

CONTAINERS

The container classes are divided into three categories, depending on the way the elements are arranged. The categories are:

Sequence Containers: They represent linear data structures, such as **deque** and

Associative Containers: Associative containers are non-linear containers that typically can locate elements stored in the containers quickly. Such containers can store sets of values or key-value pairs. The associative containers are implemented by **map** and

Derived Containers: These are constrained versions of sequential containers. They are often known as container adapters. These include **queue** and

[Table 12-1](#) highlights the specialty of each of the container classes. Apart from the containers mentioned in [Table STL](#) also provides following special containers:

Pointer-based arrays

bitsets for maintaining sets of flag values

valarrays for carrying out mathematical vector operations

Container	Description
vector	Insertion / Deletion at end. Direct access to any element.
deque	Insertion / Deletion at beginning / end. Direct access to any element.
list	Doubly Linked list. Insertion / Deletion anywhere.
set	Unique elements. Uses balanced binary search tree (BST). Fast lookups.
multiset	Duplicates allowed. Fast lookups.
map	Unique key-value pairs. Uses balanced BST. Fast key-based lookup.
multimap	Duplicates allowed. Key-value pairs. Fast key-based lookup.
stack	Last In First Out list.
queue	First In First Out list.
priority_queue	Highest priority element is first element out.

Table 12-1. Various container classes.

ITERATORS

As their name suggests, iterators allow us to go through all the items in a container and perform input/output operations on it. Iterators encapsulate the mechanism used to access container elements. Different iterators may have one or more of these capabilities. Given below is a list of iterators and their behavior.

Input iterator

- Can read an element from a container

- Can move only from beginning to end, one element at a time
- Can support only one-pass algorithms

Output iterator

- Can write an element to a container
- Can move only from beginning to end, one element at a time
- Support only one-pass algorithms

Forward iterator

- Can read/write an element from/to a container
- Can move only from beginning to end, one element at a time
- Support only one-pass algorithms

Bidirectional iterator

- Can read/write an element from/to a container
- Can move bi-directionally, one element at a time
- Can support multi-pass algorithms

Random access iterator

- Can read/write an element from/to a container
- Can move bi-directionally, skipping many elements at a time
- Can support multi-pass algorithms

Note that out of all the iterators random access iterator seems to be the most powerful. Different containers implement different types of iterators. Based on the iterator that the container supports its read/write/move capabilities get determined. Given below is a list that indicates the iterators supported by different containers.

- vector, deque - random access

- list, set, multiset, map, multimap - bidirectional
- stack, queue, priority_queue - no iterators supported

In addition to the iterator types given above, iterators can be const (e.g., “const_iterator”) or non-const. Constant iterators can be used to examine container elements, but cannot be used to modify the elements in the container. Also, non-constant iterators cannot be used with constant container objects.

ALGORITHMS

STL algorithms are template functions that perform common operations like insertion, deletion, searching, sorting and comparing elements or entire containers. As you can see, some algorithms may modify the contents of a container (for example, and some may not (for example, STL provides many algorithms. Using them can save a lot of time and effort.

Algorithms use iterators to access container elements. Algorithms often return iterators that indicate the results of the algorithms. Each algorithm has minimum requirements for the types of iterators that can be used with it.

Whether a container can be used with a specific algorithm depends upon the type of iterator that the container supports. Containers that support random-access iterators can be used with all algorithms in STL.

Since algorithms are provided as standalone functions and not as part of container classes, it is also possible to design our own algorithms and make them work on multiple container types.

Having understood the different components of STL, let us now write programs that make use of these components.

vector Container

The **vector** container resembles a C++ array in that it holds objects of the same type, and that each of these objects can be accessed individually. The **vector** container is defined as a template class, and hence it can be used to hold objects of any type. For example,

```
vector vec_int; // can hold ints  
vector vec_obj; // can hold Sample objects
```

Vectors are smarter than C++ arrays, as they can grow dynamically, as we add elements to it. **vector** always occupies contiguous memory locations. The iterator provided for the vector is a random access iterator. Let us now see a program that performs various operations on a vector.

```
#include  
#include  
using namespace std;  
  
int main()  
{
```

```

vector v1;
vector ::iterator itr1;

for (int i = 0; i <= 5; ++i)
v1.push_back (5 * i);

cout << "Vector v1:" << endl;
for (itr1 = v1.begin(); itr1 != v1.end(); ++itr1)
cout << *itr1 << endl;

vector v2 {10, 20, 30, 40, 50};
cout << "Vector v2:" << endl;
for (auto itr2 = v2.begin(); itr2 != v2.end(); ++itr2)
cout << *itr2 << endl;

vector v3 {10, 20, 30, 40, 50};
cout << "Vector v3:" << endl;
for (auto num: v3)
cout << num << endl;

return 0;
}

```

This program creates 3 vector of integers. Of these, we have stored 5 integers into **v1** by calling the function whereas, we

have initialized **v2** and **v3** in place. Then we have walked through these vectors using 3 **for** loops, each increasingly simpler than the former.

In the first **for** loop, we have called **begin()** to obtain an iterator object. The loop continues as long as **itr1** has not reached the end of the vector. This is determined by comparing **itr1** to the result of **end()** which returns an iterator indicating the location past the last element of the vector. If **itr1** is equal to this value, the end of the vector has been reached. In the body of the loop we have used ***itr1** to get the value in the current element of the vector. The expression **itr1++** positions the iterator to the next element of the vector.

In the second **for** loop, we do have used the keyword **auto**. As a result, we do not have to define the type of **It** it is deduced by the compiler from the context in which it is being used.

The third **for** loop is the new type of **for** loop. Each time through the loop, **num** will take the next value in the vector.

OTHER OPERATIONS

Apart from storing elements in a vector and walking through them, there are many other operations that can be performed

on a vector. These are shown in the following code snippet, along with appropriate comments.

```
vector v;
// store elements
v.push_back (18);
v.push_back (29);
v.push_back (-4);
v.push_back (12);
v.push_back (44);
// access elements
cout << "Element at front:" << v.front() << endl;
cout << "Element at 2nd position:" << v.at (2) << endl;
cout << "Element at end:" << v.back() << endl;

// replace elements
vector ::iterator itr;
itr = v.begin();
*itr = 35;
vec [2] = 20;
itr += 4;
*itr = 99;

// insert an element
itr = v.begin();

v.insert (itr, 25);
```

```
// erase an element
itr = v.begin();
itr += 2;
v.erase (vitr);

// remove elements from the vector
v.pop_back();

v.pop_back();

// size and capacity
cout << v.size() << endl; // current no. of elements in the
vector
cout << v.max_size() << endl; // max no. of elements vector
can hold
cout << v.capacity() << endl; // no. of ele. before a resize is
needed
v.resize (100); // increase size to 100

// purge vector elements
v.clear();
// check if vector is empty
bool b = v.empty();
```

Vector of Point Objects

Using a **vector** we can also maintain a collection of user-defined **Point** objects. The following program shows how this can be done.

```
#include
#include
using namespace std;

class Point
{
private:
int x, y;
public:
Point (int xx = 0, int yy = 0)
{
x = xx; y = yy;
}
friend ostream & operator << (ostream &o, Point &p);
};
ostream & operator << (ostream &o, Point &p)
{
o << "("<< p.x << ", "<< p.y << ")" << endl;
```

```
return o;
}

int main()
{
vector vp1;

for (int i = 0; i < 5; i++)
vp1.push_back (Point (i + 1, i + 1));

cout << "Vector vp1:" << endl;

for (auto itr : vp1)
cout << itr;

cout << "Front: ";
cout << vp1.front();
cout << "Back:";
cout << vp1.back();

vector ::reverse_iterator ritr;
cout << "Reverse Vector vp1:" << endl;
for (ritr = vp1.rbegin(); ritr != vp1.rend(); ritr++)
cout << *ritr;
cout << "Sizeof vp1:" << vp1.size() << endl;
```

```
vector vp2;
vp2.assign (vp1.begin(), vp1.begin() + 3);

cout << "Vector vp2:" << endl;
for (auto itr : vp2)
cout << itr;
}
```

Here is the output of the program...

Vector vp1:

(1, 1)
(2, 2)
(3, 3)
(4, 4)
(5, 5)

Front: (1, 1)

Back: (5, 5)

Reverse Vector vp1:

(5, 5)
(4, 4)

(3, 3)
(2, 2)
(1, 1)

Sizeof vp1: 5

Vector vp2:

```
(1, 1)  
(2, 2)
```

In this program we have first declared a **P oint** class containing a constructor and a **friend** function that displays a **Point** object. Then in **main()** in a **for** loop we have created 5 **Point** objects and added them to the vector **vp** by calling the **push_back()** function of the vector class. Next, we have iterated through the vector, using the iterator, printing values of all **Point** objects present in the vector. We have also accessed the elements at the beginning and at the end of the vector using the methods **front()** and **back()**. A reverse traversal has been done using the **size** of the container has been reported using the **size()** method. At the end, we have created another vector **vp2** and copied first 3 elements of **vp1** into it and then iterated through **vp2** from beginning to end.

list Container

list container implements a classic doubly linked list data structure. Unlike C++ array or STL vector, list elements cannot be accessed randomly. **list** is implemented as doubly linked list in order to support bi-directional iterators. Each element in the **list** contains a pointer to the preceding and the next element in the list. Lists should be used when we need frequent insertion or removal of elements from the middle of the list. Let us now see a program that performs different operations on a list container.

```
#include  
#include  
using namespace std;  
  
int main()  
{  
list ls;  
  
// add elements to the list  
ls.push_back (12);  
ls.push_front (34);
```

```
ls.push_back (19);
ls.push_front (44);
ls.push_back (31);
ls.push_front (2);
ls.push_back (29);
ls.push_front (-8);

// display elements in the list
cout << "List elements:" << endl;
for (auto litr: ls)
    cout << litr << "";
cout << endl;

cout << "Element at front:" << ls.front() << endl;
cout << "Element at end:" << ls.back() << endl;

// remove 2 elements from front and end
ls.pop_back();
ls.pop_back();
ls.pop_front();
ls.pop_front();

// display elements of the list
cout << "List after deletions:" << endl;
for (auto litr : ls)
    cout << litr << "";
```

```
cout << endl;

// insert elements
list::iterator litr;
litr = ls.end();
litr--;
ls.insert (litr, -20);
litr--;
litr--;
ls.insert (litr, 67);
litr++;
ls.insert (litr, 33);

cout << "List after insertions:" << endl;
for (auto litr : ls)
cout << litr << "";
cout << endl;

// erase an element
litr = ls.begin();
ls.erase (litr);

cout << "List after erasing 1st element:" << endl;
for (auto litr : ls)
cout << litr << "";
cout << endl;
```

```

// reverse the elements in the list
ls.reverse();
cout << "Reversed list:" << endl;
for (auto litr : ls)
cout << litr << "";
cout << endl;

// sort the elements in the list
ls.sort();
cout << "Sorted list:" << endl;
for (auto litr : ls)
cout << litr << "";
cout << endl;

// sort list in descending order
ls.sort (greater ());
cout << "Reverse Sorted list:" << endl;
for (auto litr : ls)
cout << litr << "";
cout << endl;

ls.clear();
}

```

In this program we wish to create a linked list of integers. We have called **push_back()** and **push_front()** to add elements at

the end/beginning of the list. After additions functions **front()** and **back()** display element at the first and last position of the list respectively. The functions **pop_back()** and **pop_front()** called next, remove element at the end/beginning of the list.

list has a bi-directional iterator. Random access to the elements in a list is not possible. This is the reason why **list** does not support subscript operator. Thus to insert an element at the 6th position (in a list of 7 elements), we have used the statement **litr = ls.end();** which returns an iterator referring to the element at the position. Next, we have decremented the value of iterator by 1, to make **litr** refer to the element at the position. The **insert()** function then adds element -20 at this position. Similar steps are carried out to insert element at the 3rd and position.

The **erase()** function erases an element to which the iterator **litr** is referring to. The **clear()** function on the other hand removes all the elements from the list. All elements in the list can be reversed by calling the **reverse()** function.

The **sort** algorithm function does not work on a **list**. Hence a **sort()** member function is provided for the **list** container to sort the list in ascending/descending order.

Sets and Multi-sets

These are associative containers of which a set contains unique values, whereas a multiset permits duplicates. The size of these containers can vary at runtime. They are always maintained in ascending or descending order. Both of them support bi-directional iterators. Let us now see how a set and a multiset can be used in a program.

```
#include  
#include  
#include  
using namespace std;  
  
int main()  
{  
set s1 {1, 12, -13, 44, 15, 6};  
setless > s2 {11, 32, -3, 14, 5, 12};  
setgreater > s3 {-3, 2, 4, 19, 11, 30};  
setless > s4, s5, s6, s7;  
  
cout << "s1:";  
for (auto sitr : s1)
```

```
cout << sitr << "";
cout << endl;

cout <<"s2:";
for (auto sitr : s2)
cout << sitr << "";
cout << endl;

cout <<"s3:";
for (auto sitr : s3)
cout << sitr << "";
cout << endl;

set::iterator sitr;

// search for an element
int num;
cout << "Input a number to be search:";
cin >> num;
sitr = find (s1.begin(), s1.end(), num);
if (sitr != s1.end())
cout << num << "found in s1" << endl;
else
cout << num << "not found in s1" << endl;
```

```
// check if sets set1 and set2 contains same data
bool b = includes (s1.begin(), s1.end(), s2.begin(), s2.end());
if (b)
    cout << "s1 and s2 are identical" << endl;
else
    cout << "s1 and s2 are not identical" << endl;

// get the union of two sets
set_union (s1.begin(), s1.end(), s2.begin(), s2.end(), inserter (s4,
s4.begin()));
cout << "Union of s1 and s2:" << endl;
for (auto sitr : s4)
    cout << sitr << "";
    cout << endl;

// get common elements of two sets
set_intersection (s1.begin(), s1.end(), s2.begin(), s2.end(),
inserter (s5, s5.begin()));
cout << "Intersection of s1 and s2:" << endl;
for (auto sitr : s5)
    cout << sitr << "";
    cout << endl;

// get elements of s1 not present in s2
set_difference (s1.begin(), s1.end(), s2.begin(), s2.end(), inserter
(s6, s6.begin()));
```

```
cout << "difference between sets s1 and s2:" << endl;
for (auto sitr : s6)
    cout << sitr << "";
cout << endl;

// get elements of s1 as well as s2 not present in either
set_symmetric_difference (s1.begin(), s1.end(), s2.begin(),
s2.end(), inserter (s7, s7.begin()));
cout << "symmetric difference between s1 and s2:" << endl;
for (auto sitr : s7)
    cout << sitr << "";
cout << endl;

multiset greater > ms1 {2, 3, 11, -6, 11, 2};
multiset greater > ms2 {8, 10, 11, 8, 3, 12};
multiset greater > ms3;
multiset ::iterator msitr;

// display elements of multisets
cout << "ms1:";
for (auto msitr : ms1)
    cout << msitr << "";
cout << endl;

cout << "ms2:";
for (auto msitr : ms2)
    cout << msitr << "";
```

```
cout << endl;

// get the union of two multisets

set_union (ms1.begin(), ms1.end(), ms2.begin(), ms2.end(),
inserter (ms3, ms3.begin()));

cout << "Union of ms1 and ms2:" << endl;
for (auto msitr : ms3)
cout << msitr << "";
cout << endl;
}
```

Here is the output of the program...

```
s1: -13 1 6 12 15 44
s2: -3 5 11 12 14 32
s3: 30 19 11 4 2 -3
Input a number to be search: 67
67 not found in s1
s1 and s2 are not identical
Union of s1 and s2:
-13 -3 1 5 6 11 12 14 15 32 44
Intersection of s1 and s2:
12
difference between sets s1 and s2:
-13 1 6 15 44
symmetric difference between s1 and s2:
```

```
-13 -3 1 5 6 11 14 15 32 44  
ms1: 11 11 3 2 2 -6  
ms2: 12 11 10 8 8 3  
Union of ms1 and ms2:  
12 11 11 11 10 8 8 3 3 2 2 -6
```

In this program, to begin with we have created sets **s2** and **s3** to hold integers. The functions **less** or **greater** used in the definition of **s2** and **s3** are predefined algorithm functions that ensure that elements in the set are maintained in ascending/descending order. The default order is ascending, as in the case of

The sets are traversed using the implicit iterator. Its type is deduced from the context. Function **find()** has been used to search an element in the set, whereas function **includes()** has been used to check whether two sets are identical or not.

Sets **s6** and **s7** are empty during declaration. Results of different operations on sets **s1** and **s2** are later stored in them. The operations carried out are as follows:

- Gives union of elements in **s1** and **s2**
- Gives intersection of elements in **s1** and **s2**
- Gives elements **s1** that are not present in **s2**
- Gives elements of **s1** not present in **s2** and vice versa

The template function **inserter()** copies the result of the operations performed by the above functions into the set passed to it as an argument.

In this program we have also used a multiset of integers, which can hold duplicate or multiple key values. The functions etc. works on a multiset in the same manner as they work on a set.

Maps and Multi-maps

A **map** stores a pair of values, where the pair consists of a key object and the value object. The key object can be a data such as **float** or any other object of user-defined class. The key object contains a key for which the map can be searched for. The value object stores additional data. The value object usually stores numbers or strings but it can even store the objects of other classes. For example, if the key object in a map holds a word, then the value object could be the length of the word, or the number of times the word has been repeated, or even the meaning of the word.

The data in a map always gets stored in a sorted order of the key object. The order of arranging data is decided by the function while creating the map. A map always stores a unique pair of key and value. A **multimap** on the other hand stores multiple pairs of key and value in a sorted order. Let us now see how to create a **map** and **multimap** in a program.

```
#include <iostream>
#include <map>
#include <string>
using namespace std ;

int main( )
{
    map <string, int> m1 {
        { "Rahul", 645 },
        { "Aditi", 555 },
        { "Salil", 455 },
        { "Vibha", 470 },
        { "Beena", 378 }
    };
    // display the data
    cout << "Total elements in m1: " << m1.size( ) << endl ;
    cout << "Elements in map m1: " << endl ;
    for ( auto mitr : m1 )
        cout << "Name: " << mitr.first
            << " Marks: " << mitr.second << endl ;

    // add new key-value pair
    m1 [ "Dinesh" ] = 333 ;

    // change value of existing key
    m1 [ "Beena" ] = 555 ;
```

```

// one more way to add key-value pair
pair<string, int> p ;
p.first = "Shailesh" ;
p.second = 665 ;
m1.insert ( p ) ;

cout << "New map" << endl ;
for ( auto mitr : m1 )
    cout << "Name: " << mitr.first
        << " Marks: " << mitr.second << endl ;

// search map for the given student
string str ;
cout << "Enter student name to be searched: " ;
cin >> str ;
map <string, int>::iterator mitr ;
mitr = m1.find ( str ) ;
if ( mitr != m1.end( ) )
    cout << mitr->first << " has scored "
        << mitr->second << " marks" << endl ;
else
    cout << "No such student exists!" << endl ;

multimap <string, int> m2 ;
copy ( m1.begin( ), m1.end( ), inserter ( m2, m2.begin( ) ) );
p.first = "Dinesh" ;
p.second = 665 ;
m2.insert ( p ) ;

cout << "Multi-map:" << endl ;
for ( auto mitr : m2 )
    cout << "Name: " << mitr.first
        << " Marks: " << mitr.second << endl ;
}

```

In this program we have created a map to store student's name (key) and marks (value). The elements in the map are

stored in alphabetical order by name.

We have displayed elements stored in **m1** through a loop using the implicit iterator. The statement used to display data is slightly different.

We have to use **mitr.first** to display the key value and **mitr.second** to display the associated value.

There are two ways to add new key-value pairs to a map—using the [] operator or by calling the **insert()** function and passing a key-value pair to it. We can use the [] operator to change an existing value too.

The **find()** functions is used to search the map for a specified key. If the key is found then we have displayed the marks obtained by the student, otherwise an appropriate message is displayed.

In the same program we have created a multimap Then we have copied all elements of **m1** into **m2** by calling **copy()** function. A **multimap** does not support the **operator** Hence, to add elements to the multimap we have created an object **p** of the template class Then we have called the **insert()** function to add the pair **p** to the multimap Note that the pair added is a duplicate entry, which a multimap supports. Lastly, through a **for** loop, we have displayed the elements stored in

stack Container

A **stack** object is a sequential container that allows insertion and deletion of elements only at one end. It follows Last In First Out system for adding and retrieving the stack elements. Here is the program for maintains a stack of integers.

```
#include  
#include  
using namespace std;  
  
void main()  
{  
stack stk;  
  
// add elements to the stack  
stk.push (16);  
stk.push (10);  
stk.push (19);  
stk.push (-3);  
stk.push (22);  
stk.push (18);  
int sz = stk.size();
```

```
cout << "The stack contains" << sz << "elements" << endl;

// remove elements from the stack
while (!stk.empty())
{
    int i = stk.top();
    cout << i << "";
    stk.pop();
}
```

In the program we have created a stack of integers. Once the object **stk** is created, using **push()** function we have added elements to the stack. The **size()** function returns the total number of elements present in the stack. The **top()** function returns an element present at the top of the stack. Hence, we have called this function through a **while** loop that runs till the stack does not become empty. This we have checked using **empty()** function. After displaying the element at the top, we have called **pop()** to remove an element at the top of the stack.

A stack can be implemented using vector, list or deque. We can specify the type of the underlying container as the second parameter in the constructor of **stack** as shown below:

```
stackvector> stk;
```

The default value is the class

queue Container

Like stack, we can implement a queue in our program. For example, if we wish to implement a queue of integers as a linked list then the statement to declare such a queue would be as given below:

```
queue list> q;
```

where **q** is a container of type **queue** employed as a linked list of integers. The functions like etc. work in the same manner as in Hence, I would leave it to you to write a program to maintain a queue of integers.

Suppose there are some jobs that are to be processed by the CPU. Each job has a name and a priority number. Lower the priority number, higher is that job's priority. These jobs should be presented to the CPU in the order of their priority. This can be done using the **priority_queue** container, as shown below.

```
#include  
#include
```

```
#include
#include
using namespace std;

class Task
{
private:
    string pname;
    int ppri;
public:
    Task (string n, int pr)
    {
        pname = n;
        ppri = pr;
    }
    friend class PrioritizeTasks;
    friend ostream& operator << (ostream &s, Task &t);
};

ostream& operator << (ostream &o, Task &t)
{
    o << "Process:" << t.pname << "Priority:" << t.ppri << endl;
    return (o);
}

class PrioritizeTasks
{
```

```
public:  
int operator() (const Task &t1, const Task &t2)  
{  
return t1.ppri > t2.ppri;  
}  
};
```

```
int main()  
{  
priority_queue<Task, vector<Task>, PrioritizeTasks> pq;  
Task tarr[] = {  
Task (“SWAP”, 4), Task (“PRNT”, 17),  
Task (“WORD”, 5), Task (“COPY”, 3),  
Task (“RENM”, 6), Task (“DELT”, 18),  
Task (“CRET”, 1), Task (“DUMP”, 9)  
};  
for (auto t: tarr)  
pq.push (t);
```

```
Task tk;  
while (!pq.empty())  
{  
tk = pq.top();  
cout << tk;  
pq.pop();  
}
```

```
}
```

In this program we have constructed a **priority_queue** object **pq** using the statement

```
priority_queue<vector<PrioritizeTasks>> pq;
```

pq holds **Task** objects of class **Task** as a vector. Furthermore, it states that the class **PrioritizeTasks** will provide a function to decide the order in which the tasks should get placed in the priority queue.

Once array of **Task** objects **tarr** is created, through a **for** loop we have added these objects to the priority queue **pq**, by calling function **push()**. While adding objects to **pq** an overloaded operator function **()** (which is a member function of **get**) gets called. This function compares the priorities of the two objects **t1** and **t2** and returns 0 or 1. Note that **PrioritizeClass** has been marked as a **friend** of class since the overloaded operator function **()** has to access private members of the **Task** class.

Once the priority queue is built, we have displayed its contents through a loop, calling **top()** to access individual element and **pop()** to remove it from the queue.

FUNCTION OBJECT

A function object, or functor, is a class that implements the overloaded **operator So PrioritizeTask** in our program is a functor. It is also known as a call operator. Functors are better than a function call in two respects. These are

It can contain data members.

It can be used as a template parameter.

One use of these function objects is as a sorting criterion for containers. For example, in the following declaration

```
multiset less > ms1 {2, 3, 11, -6, 11, 2};
```

less is a functor. It returns **true** if the first parameter is less than the second parameter. Since the **set** container maintains its elements in sorted order, it needs a way of comparing two elements. The comparison is done by using the functor

If we wish we can define our own sorting criteria by creating a functor and specifying it in the template list for the container as shown below.

```
multiset myfunctor > ms1 {2, 3, 11, -6, 11, 2};
```

Another use of functors is in algorithms. For example, consider the following use of **remove_if()** algorithm:

```
remove_if (v.begin(), v.end(), IsOdd);
```

Here **IsOdd** is a functor. If the element in the vector **v** is found to be odd, it would be removed from the vector. Given below is the complete implementation of this.

```
#include
#include
#include
using namespace std;

class IsOdd
{
public:
    bool operator () (int num)
    {
        return ((num % 2) == 1);
    }
};
```

```
int main()
{
vector v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
vector ::iterator pend;
vector ::iterator q;

pend = remove_if (v.begin(), v.end(), IsOdd());
for (q = v.begin(); q != pend; ++q)
cout << *q << endl;
}
```

Exercise

[A] State whether the following statements are true or false:

STL is a standard programming language.

Algorithms in STL are used for creating collections of standard and user-defined data types.

push_back() function is used to delete an element at the end of the vector.

stack and **queue** are container adapters.

set and **multiset** are types of associative containers.

[B] Answer the following:

Write a program to count the number of times a word occurs in a file.

Write a program to maintain a stack of strings.

Write a program to maintain a telephone book containing names and telephone numbers. It should be possible to search the telephone book either using a name or using a telephone number.

Write a program that applies the sort algorithm to a set of strings maintained in a vector.

KanNotes

To store, retrieve and manipulate multiple numbers / strings arrays can be used

Arrays suffer from 2 limitations:

- They have no mechanism to maintain data in different ways like key -value pairs, ordered sets, etc.
- Arrays have no means to access data in FIFO, LIFO, Sorted order, etc.

Instead of arrays we should use ready-made library called Standard Template Library (STL)

Advantages of using STL

- Very efficient, time tested, written by experts
- Readymade classes for most data structures, so we can concentrate on program rather than building data structures

- It is possible to extend the classes to suit our needs

Three key components of STL:

- 1) Containers - Store data
- 2) Iterators - Traverse container elements
- 3) Algorithms - Perform multiple opns on container elements

Container types:

Sequence - vector, deque, list

Associative - set, multiset, map, multimap

Container Adapters - stack, queue, priority_queue

Others - bitsets, valarrays

Iterator types:

Input iterator

Output iterator

Forward iterator

Bidirectional iterator

Random access iterator

Different containers support different types of iterators

Const iterators can modify elements of a container, non-const iterators cannot

Algorithms - Template functions that perform common operations like insertion, deletion, searching, sorting and comparing elements or entire containers.

A function object, or functor, is a class that implements the overloaded operator () .

[Index](#)

Some people like help, some people don't. But everybody likes "quick" help. This index will help you jump to that quick help.

A

abstract

class, [164](#)

allocation

dynamic memory, [107](#)

of arrays, [108](#)

of objects, [110](#)

of structures, [108](#)

anonymous

[25](#)

[25](#)

argument

constructor, [80](#)

default, [56](#)

assignment operator

overloading, [117](#)

[67](#)

B

[220](#)

base class, [137](#)

[202](#)

[202](#)

[190](#)

[202](#)

[190](#)

[190](#)

binding, [165](#)

dynamic, [165](#)

early, [165](#)

function, [165](#)

late, [165](#)

static, [165](#)

bool data type, [40](#)

C

C, [3](#)

C++, [7](#)

Keywords, [19](#)

Origin, [7](#)

advanced features, [229](#)

exception handling in, [293](#)

streams, [188](#)

typecasting, [250](#)

Cast

[255](#)

[252](#)

new syntax, [26](#)

[255](#)

[252](#)

void pointer, [26](#)

[293](#)

[189](#)

Classes, [79](#)

Complex, [82](#)

abstract, [164](#)

and structures, [90](#)

base, [137](#)

container,

derived, [137](#)

[233](#)

hierarchy, [150](#)

intricacies, [105](#)

linked list, [281](#)

[277](#)

virtual base, [180](#)

comments, [19](#)

Containership, [231](#)

container

dynamic, [310](#)

[326](#)

maps and multi-maps, [332](#)

[336](#)

sets and multi-sets, [329](#)

[335](#)

[321](#)

[115](#)

member functions, [40](#)

multipurpose, [115](#)

pointers, [35](#)
references, [36](#)
returning, [39](#)
[255](#)
constructors, [79](#)
arguments, [79](#)
copy, [117](#)
default, [81](#)
implicit, [81](#)
in inheritance, [147](#)

conversion routine, [124](#)
in destination object, [127](#)
in source object, [125](#)
copy constructor, [117](#)
[189](#).

D

Data
conversion, [121](#)
hiding, [10](#)
member, [76](#)
type, [40](#)
default,
argument, [56](#)
constructor, [81](#)
[109](#).
derived classes, [137](#)

destructors, [177](#)
calling virtual function, from, [180](#)
dynamic,
binding, [165](#)
containers, [310](#)
memory allocation, [107](#)
[252](#)

E

early binding, [165](#)
encapsulation, [10](#)
[22](#)
[24](#).

anonymous, [25](#)
[220](#)
[220](#)
exception,
classes, ready-made, [296](#)
classes, user-defined, [303](#)
handling, [291](#)
handling tips, [306](#)
specification, [308](#)
throwing, [297](#)
unhandled, [308](#)
extraction operator, [20](#)
[239](#).

F

[220](#)

[220](#)

[196](#)

file,

I/O, [202](#)

opening modes, [211](#)

filesystem, [221](#)

flag

error, [220](#)

flexible declarations, [23](#)

flexible initializtions, [23](#)

free store, [107](#),

[235](#)

classes, [233](#)

function, [67](#)

function, use of, [235](#)

function,

binding, [165](#)

call by address, [30](#)

call by reference, [31](#)

call by value, [30](#)

default arguments, [56](#)

definition outside the class, [92](#)

[235](#)

[65](#)

instance, [67](#)
overloading, [57](#)
prototype checking, [55](#)
pure virtual, [164](#)
[67](#)
template, [271](#)
[175](#)

G

get from, [20](#)
[204](#).
[207](#).
[220](#)

H

handling,
exception, [291](#)
heap, [107](#)
hiding data, [10](#)

I

I/O
character, [203](#)
error handling, [218](#)
expectation from, [187](#)
file, [202](#)

in C++, [20](#)

line, [206](#)

object, [215](#)

record, [207](#)

[204](#).

incremental development, [154](#).

inheritance, [137](#).

and VTABLE, VPTR, [168](#)

base class and derived class, [137](#).

constructors in, [147](#)

multiple, [152](#)

multiple levels of, [151](#)

[150](#)

[150](#)

[150](#)

uses of, [144](#).

types of, [150](#)

inferring types, [24](#)

inline functions, [64](#).

insertion operator, [20](#)

instance, [77](#)

[190](#)

[195](#)

[191](#)

[214](#).

[193](#)

K

Keyword, [19](#)

[40](#)

[293](#)

[239](#)

[241](#)

[293](#)

[295](#)

L

late binding, [165](#)

linked list, [326](#)

list container, [326](#)

M

manipulator, [195](#)

user defined, [198](#)

member function, [76](#)

arguments, [200](#)

const,

[332](#)

member, pointer to, [256](#)

memory, [88](#)

dynamic allocation, [107](#)

heap, [107](#)

static allocation, [107](#)

multi-set, [329](#)

multi-map, [332](#)

multiple
inheritance, [151](#)
levels of inheritance, [151](#)
[241](#)

N
namespaces, [242](#)
[108](#)

O
Objects, [88](#)
object oriented programming, [6](#)
characteristics, [9](#)
classes, [75](#)
inheritance, [135](#)
objects, [73](#)
polymorphism, [159](#)
reusability, [12](#)
slicing, [175](#)
stream, [189](#)
[205](#)
operator,
.*, [256](#)
::, [246](#)
++, --, [86](#)
<<, [189](#)

= and copy constructor, [117](#)

->*, [256](#)

>>, overloaded, [216](#)

assignment, [117](#)

copy constructor, [117](#)

extraction, [20](#)

insertion, [20](#)

[109](#).

overloading, [61](#)

[190](#)

[212](#)

overloading,

<<, >>, [216](#)

function, [57](#)

operator, [61](#)

operator ++, [86](#)

unary operator, [86](#)

overriding

template, [273](#)

P

pointer,

[35](#)

smart, [310](#)

[235](#)

to members, [256](#)

[27](#)

polymorphism, [159](#)
program organization, [91](#)

[77.](#)
private inheritance, [150](#)
[140](#)
protected inheritance, [150](#)

[77.](#)
pure,
virtual function, [164](#).
virtual function definition, [161](#)
put to operator, [20](#)

Q

queue container, [336](#)

R

random access, [210](#)
references, [28](#)
returning, [32](#)
[255](#)
reusability, [13](#)
RTTI, [249](#)

S

STL, [317](#)
components of, [318](#)

scope resolution operator, [246](#)

[211](#)

seeking in iostream, [211](#)

[211](#)

serialization, [217](#)

[329](#)

[309](#)

[309](#)

smart,

pointers, [310](#)

source object,

conversion routine, [125](#)

specifying exception class, [308](#)

stack container, [335](#)

static, [67](#)

binding, [165](#)

function, [67](#)

[251](#)

members, [112](#)

memory allocation, [107](#)

[21](#)

stream,

[195](#)

[191](#)

manipulators, [195](#)

[193](#)

[212](#)

streams, [188](#)
stream object, [189](#)
[93](#)
[212](#)
structure, [90](#)
structured programming, [4](#)
subobject, [180](#)
syntax
return type, [66](#)

T
[211](#)
[211](#)
template, [265](#)
application of, [287](#)

based sort, [275](#)
class, [277](#)
function, [267](#)
for user-defined types, [270](#)
linked list, [281](#)
multiple argument types, [273](#)
standard library, [317](#)
tips, [283](#)
variadic, [285](#)
versus macros, [274](#)
template override,
this pointer, [235](#)

[293](#)

throwing an exception, [293](#)

[295](#)

typecasting, [255](#)

types of inheritance, [150](#)

U

unary,

overloaded operator, [86](#)

[308](#)

unicode character printing, [194](#)

[24](#)

anonymous, [25](#)

upcast, [177](#)

using a namespace, [246](#)

V

variadic template, [285](#)

vector, [321](#)

virtual

base class, [180](#)

destructor, [177](#)

keyword, [162](#)

virtual functions, [161](#)

destructor, [177](#)

pure, [164](#)

under the hood, [166](#)

why use, [175](#)

casting, [26](#)

pointers, [27](#)

VPTR, [168](#)

VTABLE, [168](#)

w

[189.](#)

[189.](#)

[189.](#)

[189.](#)